Titre: Title:	Analyse d'heuristiques de recherche locale pour l'ordonnancement linéaire
Auteur: Author:	Benjamin Correal
Date:	2015
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Correal, B. (2015). Analyse d'heuristiques de recherche locale pour l'ordonnancement linéaire [Master's thesis, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/2040/

URL de PolyPublie: PolyPublie URL:	https://publications.polymtl.ca/2040/
Directeurs de recherche: Advisors:	Philippe Galinier
Programme: Program:	Génie informatique

UNIVERSITÉ DE MONTRÉAL

ANALYSE D'HEURISTIQUES DE RECHERCHE LOCALE POUR L'ORDONNANCEMENT LINÉAIRE

BENJAMIN CORREAL
DÉPARTEMENT DE GÉNIE INFORMATIQUE ET GÉNIE LOGICIEL
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION
DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES
(GÉNIE INFORMATIQUE)
NOVEMBRE 2015

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé :

ANALYSE D'HEURISTIQUES DE RECHERCHE LOCALE POUR L'ORDONNANCEMENT LINÉAIRE

présenté par : CORREAL Benjamin
en vue de l'obtention du diplôme de : Maîtrise ès sciences appliquées
a été dûment accepté par le jury d'examen constitué de :

M. DESMARAIS Michel C., Ph. D., président
M. GALINIER Philippe, Doctorat, membre et directeur de recherche
M. PESANT Gilles, Ph. D., membre

REMERCIEMENTS

J'aimerais d'abord remercier mon directeur de recherche, Philippe Galinier, pour son aide précieuse et ses conseils tout au long de la recherche. Je tiens aussi à remercier ma famille et mes amis pour tout leur soutien dans mon cheminement.

RÉSUMÉ

Ce mémoire porte sur le problème d'ordonnancement linéaire (LOP – *Linear Ordering Problem*) et le problème équivalent de *Feedback Arc Set* (FASP). Ces deux problèmes NP-difficiles se distinguent surtout par le type d'exemplaires traités et ont chacun plusieurs applications pratiques dans divers domaines. Alors que le LOP est bien étudié dans la littérature, très peu d'heuristiques sont proposées pour le FASP. Les heuristiques les plus efficaces pour traiter ces problèmes, comme l'algorithme mémétique et la recherche locale itérée, sont des heuristiques hybrides et utilisent à répétition la recherche locale sous forme de descentes. L'implémentation d'un opérateur de recherche est donc critique à la performance de ces heuristiques. L'objectif de ce travail est donc d'implémenter des heuristiques de recherche locale de façon efficace pour le LOP et pour le FASP.

Dans ce travail, nous décrivons les heuristiques appliquées au LOP dans la littérature et particulièrement les techniques existantes pour effectuer la recherche locale. Les méthodes les plus efficaces sont alors implémentées et plusieurs améliorations sont proposées pour accélérer le traitement des exemplaires de grande taille ou de faible densité. Par la suite, nous présentons une étude expérimentale des divers opérateurs de recherche en comparant leurs temps d'exécutions pour effectuer une descente sur des exemplaires générés aléatoirement de taille et densité variables. De plus, un modèle simple est utilisé pour décomposer le temps d'exécution en plusieurs facteurs afin d'analyser en détail l'impact de l'implémentation d'un opérateur et de la politique de recherche appliquée.

Nous observons que les opérateurs de recherche développés sont très efficaces dans différentes situations et permettent d'améliorer les méthodes existantes. Nous proposons alors des recommandations pour sélectionner l'opérateur approprié selon le type d'exemplaire du problème à traiter en plus de donner des conseils pour son implémentation. En particulier, nous présentons le meilleur opérateur à utiliser pour des exemplaires FASP alors que ce sujet n'a pas été abordé auparavant. Les divers opérateurs de recherche décrits pourront alors être incorporés dans une heuristique hybride afin d'accélérer son exécution.

ABSTRACT

This thesis focuses on the *Linear Ordering Problem* (LOP) and the equivalent *Feedback Arc Set Problem* (FASP). Both NP-hard problems differ mainly by the type of instances they handle and they each have several practical applications in various fields. While the LOP is much studied in the literature, very few heuristics are proposed for the FASP. The most efficient heuristics proposed to tackle these problems, such as memetic algorithm and iterated local search, are hybrid heuristics and use local search repeatedly in a hill climbing (HC) operator. The implementation of a HC operator is critical to the performance of these heuristics. The objective of this work is to implement local search heuristics efficiently for the LOP and the FASP.

In this work we describe the heuristics applied to the LOP in the literature and focus on existing techniques to perform local search. The most efficient methods are then implemented and several improvements are proposed to speed up the execution for large or sparse instances. Thereafter, we present an extensive experimental study of the different variants of the HC operator by comparing their running times to reach a local optimum on a large set of randomly generated problem instances which have various sizes and densities. In addition, a simple model is used to decompose the running time into several factors to analyze in detail the impact of the implementation of an operator and the search policy.

We observe that the proposed variants of the HC operator are very efficient in different situations and can improve existing methods. We then present recommendations to select an appropriate HC operator according to the type of problem instances to tackle. We also provide some advice for its implementation. Notably, we present the best HC operator to solve FASP instances. This issue has not been addressed before. The various HC operators described can then be incorporated into an hybrid heuristic to speed up its execution.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

LISTE DES SIGLES ET ABRÉVIATIONS

BI	Politique de la meilleure amélioration – *Best Improvement*
CITO	Problème d'ordre de test d'intégration de classe – *Class Integration Test Order*
FASP	*Feedback Arc Set Problem*
FI	Politique de la première amélioration – *First Improvement*
FVI	Politique de la première amélioration d'élément – *First Variable Improvement*
GA	Algorithme génétique – *Genetic Algorithm*
HC	Algorithme de descente – *Hill Climbing*
HCR	Descente à partir d'une configuration aléatoire
ILS	Recherche locale itérée – *Iterated Local Search*
LOP	Problème d'ordonnancement linéaire – *Linear Ordering Problem*
MA	Algorithme mémétique – *Memetic Algorithm*
MPS	Somme partielle maximale – *Maximum Partial Sum*
SS	Recherche dispersée – *Scatter Search*
TS	Recherche tabou – *Tabu Search*
VNS	Recherche à voisinage variable – *Variable Neighborhood Search*

LISTE DES ANNEXES

CHAPITRE 1 INTRODUCTION

La résolution de problèmes d'optimisation combinatoire pose un défi important en informatique et en mathématiques et se retrouve dans plusieurs applications dans des domaines variés. Dans ce mémoire, on s'intéresse en particulier au problème d'ordonnancement linéaire (LOP – *Linear Ordering Problem*) et au problème équivalent de *Feedback Arc Set* (FASP). Bien que ces deux problèmes soient pratiquement identiques, ils sont généralement formulés différemment. En effet, le LOP est représenté par une matrice, avec des exemplaires typiquement relativement petits et denses, alors que les exemplaires FASP correspondent plutôt à un graphe, de grande taille et peu dense. De plus, le FASP est très peu étudié dans la littérature, mais possède tout de même de nombreuses applications pratiques. On cherche donc à développer des heuristiques efficaces pour résoudre le LOP ainsi que le FASP. Puisque plusieurs heuristiques dédiées au LOP utilisent la recherche locale, son implémentation est critique à la performance d'un algorithme. On se concentre donc spécifiquement sur l'implémentation efficace de la recherche locale pour le LOP et le FASP. Dans ce chapitre, le problème LOP est expliqué en détail ainsi que quelques concepts de base. Ensuite, l'objectif de la recherche est présenté suivi d'un plan du mémoire.

1.1 Le problème d'ordonnancement linéaire

Le problème d'ordonnancement linéaire (LOP – *Linear Ordering Problem*) est un problème d'optimisation combinatoire classique dont la version décisionnelle fait partie des 21 problèmes NP-complets de Karp [28]. Le problème d'optimisation en tant que tel est montré NP-difficile par Garey et Johnson [17] en 1979.

Le LOP consiste à la base à permuter l'ensemble des éléments de 1 à n. Pour toute paire d'éléments (p, q), tel que $1 \leq p, q \leq n$ et $p \neq q$, il y a un gain C_{pq} si p est placé avant q. L'objectif est de trouver une permutation de cet ensemble qui maximise la somme des gains. Formellement, un exemplaire du problème est défini par une matrice $n \times n$ notée C. Une solution valide du problème consiste en une permutation π de $\{1, \ldots, n\}$ et son score $f(\pi)$ correspond à la Formule 1.1. L'objectif du LOP est de trouver la permutation π qui maximise $f(\pi)$.

$$f(\pi) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} C_{\pi_i \pi_j} \tag{1.1}$$

Le LOP peut aussi être représenté sous la forme d'un problème de graphe. Soit $G = (V, E)$, un graphe orienté constitué de l'ensemble des sommets V et des arcs E, et un poids w_{pq} associé aux arcs pour $p, q \in V$, il faut trouver un sous-ensemble d'arcs T tel que $\sum_{(p,q)\in T} w_{pq}$ est maximisé et $G' = (V, T)$ ne contient aucun cycle. À partir du graphe orienté acyclique G', il est possible d'obtenir un ordre topologique des sommets pour former une solution sous la forme d'une permutation comme dans la représentation matricielle du LOP. On note le nombre de sommets et d'arêtes $n = |V|$ et $m = |E|$ respectivement.

Le LOP a des applications dans plusieurs domaines. En particulier, le problème de triangulation de tableaux d'entrée-sortie dans le domaine de l'économie [11] consiste à ordonner simultanément les lignes et colonnes d'une matrice pour maximiser la somme des valeurs dans le triangle supérieur. Ce problème a une correspondance directe avec la définition matricielle énoncée ci-dessus. De plus, la représentation sous forme de graphe est associée au problème du *Feedback Arc Set* (FASP), qui consiste à trouver le plus petit sous-ensemble d'arcs dont le retrait donne un graphe orienté acyclique [23]. La solution de ce problème est alors obtenue par $E - T$. Ces deux problèmes sont totalement équivalents. Le FASP est rencontré dans la littérature dans sa version pondérée ainsi que non pondérée.

Plusieurs autres applications ont été mentionnées dans la littérature. Par exemple : l'agrégation de préférences individuelles [30] ; la relation d'ancienneté optimale en archéologie [21] ; le classement dans un tournoi de sport [23] ; la représentation planaire de graphe biparti qui minimise le croisement d'arcs [26] ; et la minimisation du temps total pondéré dans l'ordonnancement de tâches sur une seule machine [3].

Enfin, une application qui est peu mentionnée dans la littérature est de trouver l'ordre optimal des classes pour des tests d'intégration dans le développement logiciel. Il s'agit du problème d'ordre de test d'intégration de classe (CITO – *Class Integration Test Order*). Dans une situation idéale, lorsqu'une classe doit être testée, toutes ses dépendances ont été intégrées et testées auparavant. Par contre, ce n'est pas toujours possible puisqu'il peut y avoir des cycles de dépendances. Il faut donc briser ces cycles en simulant certaines classes. Le problème *CITO* consiste alors à trouver un ordonnancement des classes qui minimise le travail requis pour développer les classes simulées. Ce problème est donc équivalent au FASP et, par extension, au LOP [4].

1.2 Définitions et concepts de base

1.2.1 Problème d'optimisation NP-difficile

Un problème d'optimisation est caractérisé par une fonction objectif, qui permet d'évaluer quantitativement une solution, et certaines contraintes à respecter. Le but de l'optimisation est non seulement de trouver une solution valide, mais bien de trouver la meilleure solution parmi toutes celles possibles, c'est-à-dire celle qui maximise (ou minimise selon la définition du problème) la valeur de la fonction objectif. Lorsqu'on mentionne le score ou la qualité d'une solution, il s'agit de la valeur de la fonction objectif. De plus, la qualification NP-difficile indique la complexité du problème. En théorie de la complexité, la classe NP peut être résolue en temps polynomial par une machine de Turing non déterministe. Par contre, un ordinateur est une machine de Turing déterministe et peut donc seulement résoudre le problème en temps exponentiel en supposant que la fameuse conjecture "$P \neq NP$" est vraie. Cela signifie que, en augmentant la taille du problème, le temps requis pour le résoudre croît très rapidement. Ainsi, il est souvent irréaliste de chercher la solution optimale même pour des exemplaires du problème relativement petits. On s'intéresse alors, dans ce cas, à obtenir une bonne solution dans un temps raisonnable à l'aide d'une heuristique.

1.2.2 Heuristique

Une heuristique est un algorithme d'optimisation qui permet de trouver rapidement une solution à un problème sans toutefois fournir de garantie sur la qualité des solutions. Ce type d'algorithme est souvent utilisé pour résoudre des problèmes NP-difficiles puisqu'on ne s'attend pas à pouvoir atteindre la solution optimale dans un délai raisonnable. Une métaheuristique est un algorithme heuristique de haut niveau qui peut être adapté à divers problèmes.

Certaines métaheuristiques maintiennent plusieurs configurations simultanément. L'ensemble de ces solutions est appelé la population. Dans ce cas, une importance est généralement accordée à conserver une certaine diversité dans les configurations. On se fie alors à la mesure de distance entre des configurations, qui varie selon le problème, mais revient souvent à approximer le nombre de mouvements nécessaires pour passer d'une configuration à une autre. Le concept de diversité est semblable à la distance entre deux configurations, mais appliqué à un ensemble de solutions.

1.2.3 Recherche locale

La recherche locale est un mécanisme qui consiste à modifier itérativement une solution existante du problème, aussi appelée configuration, dans le but d'améliorer progressivement la qualité de la solution courante. Chaque modification à la configuration courante est appelée un **mouvement** et permet de se déplacer dans l'**espace de recherche**, qui consiste en l'ensemble des solutions du problème. On utilise la notation suivante : soit une configuration π et un mouvement m, on note $\pi \oplus m$ la configuration obtenue lorsque m est appliqué à π. Le sous-ensemble de l'espace de recherche qui peut être atteint en appliquant un mouvement à une configuration donnée est appelé son **voisinage**. Lorsqu'une solution a un score meilleur ou égal à celui de tous ses voisins, on dit qu'elle est un **optimum local** puisqu'aucun mouvement ne peut l'améliorer. Un **optimum global** correspond à la solution optimale du problème, qui donne la meilleure valeur de la fonction objectif.

L'étape la plus longue dans la recherche locale est généralement l'exploration du voisinage, c'est-à-dire évaluer une partie ou tous les mouvements afin de choisir le voisin qui deviendra la prochaine configuration courante. La stratégie utilisée pour faire ce choix est appelée la **politique de recherche**. Une mesure utile est la **performance d'un mouvement**, qui consiste en l'impact sur le score d'effectuer un certain mouvement. Avec $f(\pi)$ la fonction objectif, la performance d'un mouvement est calculée par $\delta(m) = f(\pi \oplus m) - f(\pi)$. Dans le cas d'un problème de maximisation comme le LOP, une valeur $\delta(m)$ positive indique que le mouvement améliore la configuration courante. Il est tout de même possible, et parfois très utile, d'appliquer un mouvement qui détériore une solution pour arriver à un meilleur résultat à long terme. L'**implémentation** de l'exploration du voisinage correspond à la technique et au besoin, aux structures de données utilisées pour évaluer la performance des mouvements en accord avec la politique choisie.

Une application simple de recherche locale consiste à appliquer itérativement un mouvement qui améliore la configuration courante, avec une politique de recherche quelconque, jusqu'à ce qu'un optimum local soit atteint. Cette technique est appelée la **descente**. Il est à noter que le terme *recherche locale* est souvent employé dans la littérature pour signifier une descente, bien que la recherche locale soit un terme plus général dont la descente est un cas particulier. Dans ce document, on utilise l'appellation **opérateur de recherche** pour désigner la combinaison d'une politique de recherche, d'un voisinage et de l'implémentation de la recherche dans ce voisinage avec la politique choisie dans le cadre d'une descente.

1.2.4 Caractéristiques du LOP

Pour tout le document, les exemplaires du problème seront considérés comme normalisés. La notion d'exemplaire normalisé est défini par Martí et Reinelt [35] et requiert que, dans la représentation matricielle, toutes les valeurs de la matrice C soient des entiers non négatifs. De plus, soit C_{pq} ou C_{qp} doit être égal à 0 pour $1 \leq p \leq q \leq n$, incluant tous les éléments de la diagonale. Pour la représentation par un graphe, ces contraintes sont alors que, pour $p, q \in V$, w_{pq} est entier et non négatif et qu'un seul des arcs (p, q) ou (q, p) peut être contenu dans le graphe. Il est à noter qu'un exemplaire normalisé génère les mêmes optima que la version originale, mais le score final peut être différent.

La **densité**, simplement notée d est souvent utilisée dans ce document pour caractériser et catégoriser des exemplaires du problème. Cette mesure est plus simple à visualiser avec la représentation par graphe du problème. La densité correspond à la proportion d'arcs présents dans le graphe par rapport au nombre d'arcs possibles. Soit un graphe $G = (V, E)$, sa densité est alors $d = \dfrac{2 \times |E|}{|V| \times (|V| - 1)}$. Typiquement, pour le problème LOP, on s'intéresse à des exemplaires relativement petits de densité élevée, alors que pour le FASP, on traite plutôt de gros exemplaires de faible densité.

1.3 Objectifs de la recherche

La recherche locale a un rôle important dans plusieurs méthodes de résolution du LOP, souvent directement sous la forme d'une descente, tel que dans les algorithmes de recherche locale itérée, de recherche à voisinage variable et l'algorithme mémétique présentés à la section 2.4. L'exploration du voisinage est donc un élément critique à la performance d'un algorithme. Il existe principalement deux techniques d'implémentation dans la littérature. L'implémentation classique est conçue pour traiter des exemplaires denses et est utilisée par la plupart des algorithmes récents appliqués au LOP. L'implémentation arborescente quant à elle s'adapte à la densité des exemplaires et profite d'une complexité asymptotique plus faible dans le pire cas. Elle pourrait donc avoir un avantage important pour les gros exemplaires de faible densité. Les auteurs ayant proposé cette implémentation affirment d'ailleurs qu'elle est significativement plus rapide que ce qu'on retrouve dans la littérature. Par contre, dans l'étude réalisée par ces auteurs, l'opérateur de recherche avec l'implémentation classique n'utilise pas la politique de recherche qui est la plus efficace pour cette implémentation. De plus, il est possible d'améliorer l'implémentation classique pour l'adapter aux exemplaires peu denses. On s'interroge donc sur l'avantage de l'implémentation arborescente par rapport à l'implémentation classique pour les situations dans lesquelles la taille des exemplaires est très élevée

ainsi que lorsque la densité est très faible. Ainsi, les objectifs de la recherche sont les suivants :

— Implémenter de manière efficace des opérateurs de recherche appliquant les implémentations présentes dans la littérature avec les politiques de recherche pertinentes.
— Comparer les performances de ces opérateurs de recherche dans l'algorithme de descente.
— Guider le choix d'un opérateur de recherche selon l'utilisation requise.

1.4 Plan du mémoire

Dans ce chapitre, la problématique et plusieurs concepts de base ont été expliqués. Ensuite, une revue critique de la littérature est présentée au chapitre 2, dans laquelle les méthodes existantes pour résoudre le LOP sont détaillées et comparées. Puis, la méthodologie suivie dans ce travail est décrite au chapitre 3. La majeure partie de la recherche et des résultats est contenue dans l'article au chapitre 4. Cet article présente les opérateurs de recherche développés ainsi qu'une analyse expérimentale détaillée des divers opérateurs existants. Quelques résultats complémentaires sont ensuite présentés au chapitre 5. Par la suite, le chapitre 6 discute des résultats obtenus et de l'ensemble du travail de recherche. Enfin, au chapitre 7, une conclusion souligne les contributions du travail, identifie leurs limitations et présente des recommandations et des voies de recherche.

CHAPITRE 2 REVUE DE LITTÉRATURE

Le problème d'ordonnancement linéaire a été largement étudié depuis plus d'un demi-siècle. Ce chapitre présente une revue de littérature des méthodes permettant de traiter le LOP. D'abord, les principaux algorithmes sont expliqués en détail. Ensuite, on décrit les jeux de données communément évalués. Finalement, une comparaison identifie les techniques les plus efficaces.

2.1 Méthodes exactes

Il existe plusieurs méthodes exactes appliquées au LOP pour trouver une solution optimale. Par contre, il s'agit d'algorithmes exponentiels, donc seuls des exemplaires de petite taille ou avec des caractéristiques particulières peuvent être traités dans un temps raisonnable.

2.1.1 Branch-and-bound

La méthode branch-and-bound, décrite par Land et Doig [34], trouve une solution optimale en divisant récursivement le problème original en problèmes plus petits et en résolvant ceux-ci. C'est donc une approche *diviser pour régner*. La division du problème peut être faite de différentes façons et consiste souvent à fixer certains attributs de la solution et évaluer indépendamment chacune des valeurs possibles. Cette division récursive crée un arbre de solutions qui couvre tout l'espace de recherche. Pour éviter d'évaluer toutes les solutions possibles, les bornes inférieure et supérieure sont calculées à chaque division du problème en espérant pouvoir éliminer d'un coup de larges portions de l'espace de recherche. Lorsque la borne supérieure d'une branche de l'arbre est moindre qu'une borne inférieure déjà connue, toutes les solutions engendrées par ce branchement sont ignorées puisqu'elles ne peuvent pas contenir la solution optimale.

Un algorithme branch-and-bound appliqué au LOP est proposé par deCani [15] dans lequel la subdivision du problème est faite en ordonnant un certain nombre d'éléments de la configuration. Chaque division considère un élément de plus et l'insère à diverses positions dans la solution partielle. Une autre version est développée par Kaas [27] où la subdivision part d'une configuration dont les premiers éléments sont fixés et la complète en ajoutant, à la fin, un des éléments restants. Une méthode plus récente présentée par Charon et Hudry [10] se distingue en utilisant une relaxation lagrangienne pour calculer les bornes et trouve généralement la solution optimale pour des exemplaires de moins de 100 éléments.

2.1.2 Branch-and-cut

La méthode branch-and-cut tente de résoudre un problème d'optimisation linéaire en nombres entiers à l'aide de la méthode des plans sécants (*cutting planes*) [29] et de branch-and-bound. Le LOP est alors traité comme un problème d'optimisation linéaire. La méthode des plans sécants consiste à rajouter des contraintes au problème lorsqu'une solution trouvée contient des valeurs qui ne sont pas entières afin de continuer la recherche et espérer trouver une solution contenant uniquement des entiers. Lorsqu'aucun autre plan sécant n'est trouvé, le problème est divisé et la méthode de branch-and-bound est appliquée. La borne inférieure est calculée avec les solutions entières trouvées et la borne supérieure avec les solutions non entières pour chaque branchement. Chaque division est alors résolue en répétant ce processus. Les deux parties les plus importantes de ce type d'algorithme sont la manière de trouver des plans sécants et la méthode de résolution de problème d'optimisation linéaire.

Des implémentations de branch-and-cut ont été proposées entre autres par Grötschel et al. [23] et Christof et Reinelt [12] et se distinguent par le type de plans sécants utilisé. Un autre algorithme par Mitchell et Borchers [37] utilise plutôt une variante pour résoudre l'optimisation linéaire et est capable de trouver une solution optimale pour des exemplaires dans leur jeu de données contenant jusqu'à 250 éléments.

2.2 Méthodes de construction

Comme leur nom l'indique, les méthodes de construction servent à générer de nouvelles solutions. Elles utilisent alors un critère pour maximiser le score des solutions obtenues. Ces méthodes sont généralement des algorithmes gloutons qui s'exécutent très rapidement, mais créent des solutions qui peuvent encore être améliorées facilement. Il est donc possible de les utiliser pour générer des solutions initiales pour un algorithme plus complexe plutôt que de commencer par une configuration aléatoire.

Une idée générale est proposée par Chenery et Watanabe [11] dès 1958 indiquant quels éléments mettre au début ou à la fin de la permutation. Aujac [1] propose des coefficients qui servent à ordonner chaque paire d'éléments et une configuration est construite en tentant de satisfaire ces ordres. L'heuristique de Becker [2] présente un critère glouton qui permet de générer de bonnes solutions relativement à l'effort de calcul demandé. L'élément avec le score maximal est itérativement inséré à la fin de la configuration. Ce critère et la méthode de construction sont utilisés entre autres par [5, 42, 44]. Martí et Reinelt [35] décrivent et comparent ces heuristiques de construction ainsi que deux variantes d'une méthode par insertions où les éléments sont insérés dans un ordre aléatoire à la position qui maximise leur

contribution au score final. Leur comparaison indique que la méthode de Becker et celles par insertions génèrent des solutions de meilleure qualité.

2.3 Recherche locale

La recherche locale est un outil puissant dans l'optimisation de solution. Cette section complémente l'information sur plusieurs concepts de base de cette technique qui ont été expliqués à la section 1.2.3 et présente leurs applications au LOP.

2.3.1 Mouvement et voisinage

Le choix d'un type de mouvement est fondamental dans la recherche locale. En effet, la définition du type de mouvement définit le voisinage et les optima locaux. Cette section décrit les principaux types de mouvement mentionnés dans la littérature. Comme indiqué à la Section 1.2.3, on considère la notation $\pi \oplus m$ correspondant à la configuration obtenue lorsque le mouvement m est appliqué à la configuration π. De plus, un voisinage du même nom est associé à chacun des types de mouvement présentés. Pour une configuration donnée, son voisinage englobe toutes les configurations qu'il est possible d'atteindre en lui appliquant le type de mouvement choisi.

Le mouvement d'insertion est le plus souvent utilisé pour le LOP. Il consiste à retirer un élément π_i de la configuration et à le réinsérer à la position j. Ce mouvement sera noté $<\pi_i \rightarrow j>$, où $1 \leq i, j \leq n$ et $i \neq j$. L'équation 2.1 présente en détail l'application du mouvement d'insertion.

$$\pi \oplus <\pi_i \rightarrow j> = \begin{cases} (\pi_0, \ldots, \pi_{i-1}, \pi_{i+1}, \ldots, \pi_j, \pi_i, \pi_{j+1}, \ldots, \pi_n), & i < j \\ (\pi_0, \ldots, \pi_{j-1}, \pi_i, \pi_j, \ldots, \pi_{i-1}, \pi_{i+1}, \ldots, \pi_n), & i > j \end{cases} \tag{2.1}$$

Le mouvement d'échange (*interchange* en anglais) est aussi mentionné à quelques reprises. Il s'agit d'intervertir deux éléments π_i et π_j, noté $<\pi_i \leftrightarrow \pi_j>$, tel que décrit par l'équation 2.2.

$$\pi \oplus <\pi_i \leftrightarrow \pi_j> = \begin{cases} (\pi_0, \ldots, \pi_{i-1}, \pi_j, \pi_{i+1}, \ldots, \pi_{j-1}, \pi_i, \pi_{j+1}, \ldots, \pi_n), & i < j \\ (\pi_0, \ldots, \pi_{j-1}, \pi_i, \pi_{j+1}, \ldots, \pi_{i-1}, \pi_j, \pi_{i+1}, \ldots, \pi_n), & i > j \end{cases} \tag{2.2}$$

Un mouvement plus limité est nommé *swap*. Il correspond à déplacer un élément à une position adjacente. Il peut être défini de manière équivalente par les deux types de mouvement présentés précédemment comme $<\pi_i \rightarrow j>$ ou $<\pi_i \leftrightarrow \pi_j>$, où $j = i \pm 1$.

Le mouvement d'insertion est plus fréquemment utilisé que les mouvements d'échange et *swap,* car de multiples résultats expérimentaux affirment qu'il procure de meilleures performances au niveau du temps de calcul et de la qualité des solutions retournées [33, 35, 43]. De plus, Huang et Lim [25] démontrent qu'un mouvement d'insertion peut éventuellement améliorer un optimum local du voisinage échange, mais l'inverse n'est pas possible. Ainsi, on se concentre principalement sur le voisinage d'insertion. De plus, dans ce document, à moins d'indication contraire, toute recherche locale est implicitement appliquée avec le voisinage d'insertion.

Un voisinage supplémentaire nommé *dynasearch* est proposé par Congram [13] et est aussi basé sur le mouvement d'insertion décrit à l'équation 2.1. La notion de mouvements indépendants est définie comme suit : $<\pi_i \rightarrow j>$ et $<\pi_k \rightarrow l>$ sont considérés indépendants si $\max(i,j) \leq \min(k-1,l)$ ou $\min(i-1,j) \geq \max(k,l)$. Le voisinage *dynasearch* contient toutes les configurations qui peuvent être obtenues par des mouvements d'insertion qui sont tous indépendants deux à deux.

Le voisinage restreint d'insertion, proposé par Ceberio et al. [7], effectue une analyse préliminaire de l'exemplaire à traiter dans le but d'éliminer certains mouvements d'insertion de la recherche. Lors de cette initialisation, chaque élément de la configuration est évalué à toutes les positions avec la permutation qui lui est la plus favorable. Si, en plaçant l'élément p à la position j dans le meilleur des cas, il est tout de même possible de déplacer p pour améliorer la solution, il est conclu qu'aucun optimum local ne peut contenir p à la position j. Le mouvement $<p \rightarrow j>$ peut alors être ignoré. Ainsi, une matrice de la même taille que l'exemplaire est générée en $\mathcal{O}(n^3)$ indiquant pour chaque élément quelles positions sont acceptables. Le critère pour rejeter un mouvement permet seulement d'ignorer des positions subséquentes aux deux extrémités. En pratique, cette méthode s'avère efficace seulement lorsqu'une certaine proportion des mouvements sont interdits et il est observé que cette valeur diminue lorsque la taille des exemplaires augmente, réduisant l'efficacité de cette méthode.

2.3.2 Politique de recherche

La politique de recherche indique quelle configuration choisir lors de l'exploration du voisinage. La meilleure amélioration (BI – *Best Improvement*) et la première amélioration (FI – *First Improvement*) sont deux politiques communément rencontrées. La politique BI évalue chaque voisin de la configuration courante et choisit celui qui offre la meilleure amélioration du score courant. En général, un choix aléatoire est effectué dans le cas d'ex æquo. La politique FI parcoure le voisinage et effectue le premier mouvement qui apporte une amélioration à la configuration courante. Le voisinage est généralement exploré de façon aléatoire, mais

peut aussi être fait dans un ordre précis selon une heuristique.

Une troisième politique qui comporte des éléments de BI et FI est avancée pour le voisinage d'insertion par Laguna et al. [33]. Chaque élément de la configuration est sélectionné, un à la fois, dans un ordre déterminé. Si le meilleur mouvement à partir d'un élément sélectionné améliore le score courant, il est appliqué immédiatement et la recherche locale passe à l'itération suivante. Chaque mouvement déplace donc un élément à sa position optimale dans la configuration courante. Cette politique n'est pas nommée dans la littérature alors elle sera désignée, dans ce document, par la première amélioration d'élément (FVI – *First Variable Improvement*).

Lors de comparaisons expérimentales, il est déterminé que la politique FVI est plus rapide que BI [33] et FI [43].

2.3.3 Implémentation de l'exploration de voisinage

Appliquer un mouvement à une configuration est seulement $\mathcal{O}(n)$ dans le cas du LOP. La partie la plus coûteuse de la recherche locale est donc généralement l'exploration du voisinage, qui consiste à évaluer la performance de certains mouvements, selon la politique choisie. Dans le cas de BI, tous les mouvements sont évalués. On s'intéresse alors à la complexité d'effectuer une itération de descente avec différentes implémentations. On se concentre surtout sur le voisinage d'insertion puisque c'est celui qui est le plus utilisé dans la littérature et qui donne les meilleurs résultats.

Dans la section 1.2.3, on définit la performance d'un mouvement par $\delta(m) = f(\pi \oplus m) - f(\pi)$. On peut alors définir la performance d'un mouvement d'insertion selon l'équation 2.3. On observe qu'évaluer cette fonction est $\mathcal{O}(n)$.

$$\delta(<\pi_i \to j>) = \begin{cases} \sum_{k=i+1}^{j} C_{\pi_k \pi_i} - C_{\pi_i \pi_k}, & i < j \\ \sum_{k=j}^{i-1} C_{\pi_i \pi_k} - C_{\pi_k \pi_i}, & i > j \end{cases} \tag{2.3}$$

Afin de simplifier la notation dans les équations suivantes, on définit $D[p][q] = C_{pq} - C_{qp}$, pour tout $p, q = 1 \ldots n$. À noter que $D[p][q]$ correspond à l'avantage sur le score de placer p avant q plutôt que l'inverse. En particulier, $D[\pi_{i+1}][\pi_i] = \delta(<\pi_i \to i+1>)$. Schiavinotto et Stützle [43] suggèrent de précalculer cette matrice pour obtenir une légère optimisation.

La taille du voisinage d'insertion est $n \times (n-1)$ puisque, pour chaque élément p parmi les n dans la configuration, il existe $n-1$ positions j auxquelles il peut être déplacé. On distingue deux parties distinctes dans l'exploration du voisinage : itérer les éléments p et itérer les

positions j pour une valeur de p déterminée. Les différentes implémentations présentent des techniques pour effectuer cette deuxième partie de manière plus efficace.

Implémentation de base

L'implémentation de base consiste à calculer le score de chacun des mouvements indépendamment avec l'équation 2.3. Cette implémentation est donc $\mathcal{O}(n^3)$.

Implémentation quadratique

Une implémentation plus efficace est expliquée par Congram [13] en 2000. En évaluant les mouvements à partir d'un élément p de façon consécutive de chaque côté à partir de p, il est possible de calculer la performance de tous les mouvements en $\mathcal{O}(n)$ grâce à l'équation récursive 2.4. Il suffit donc de deux boucles successives évaluant les mouvements des deux côtés de p, totalisant $n-1$ opérations. L'évaluation d'un seul mouvement apparait alors être en temps constant et le voisinage peut être exploré en $\mathcal{O}(n^2)$.

$$\begin{aligned} \delta(<\pi_i \rightarrow i>) &= 0 \\ \delta(<\pi_i \rightarrow j>) &= \delta(<\pi_i \rightarrow j-1>) + D[\pi_j][\pi_i] \end{aligned} \tag{2.4}$$

Implémentation par listes

Deux implémentations qui, contrairement à celles présentées précédemment, s'adaptent à la densité des exemplaires sont proposées par Sakuraba et Yagiura [41]. Plutôt que d'effectuer tout le travail lors de l'exploration du voisinage pour trouver un mouvement, ces implémentations utilisent des structures de données qui doivent être mises à jour lorsqu'un mouvement est appliqué. Pour les deux implémentations, l'espace mémoire requis est $\mathcal{O}(m)$.

L'implémentation par listes est décrite dans [41] et maintient, pour chaque élément, une liste de ses voisins ordonnée en fonction de la configuration courante. Il est alors possible d'évaluer les mouvements d'un élément donné en itérant uniquement ses voisins et en ignorant les autres éléments puisqu'ils n'affectent pas la performance des mouvements. L'initialisation de ces listes et l'exploration du voisinage ont alors une complexité $\mathcal{O}(m)$. Lorsqu'un mouvement $<p \rightarrow j>$ est appliqué, il suffit de mettre à jour les listes qui contiennent l'élément p, c'est-à-dire les listes correspondant à p et ses voisins. Cette mise à jour est aussi appliquée en $\mathcal{O}(m)$, ce qui est donc aussi la complexité d'une itération de recherche locale.

Implémentation arborescente originale

L'implémentation arborescente originale, la deuxième décrite dans [41], raffine celle par listes en utilisant un arbre équilibré pour chaque élément plutôt qu'une liste. L'arbre associé à un élément p représente dans ses feuilles l'intervalle entre chacun des voisins de p ordonnés selon la configuration courante, mais pas p lui-même. Chaque nœud de l'arbre contient, entre autres, une valeur γ, qui est utilisée pour les calculs internes et n'a pas de signification particulière, ainsi que $\gamma_{\max}$ qui correspond à la somme maximale des γ sur tous les chemins reliant le nœud à une de ses feuilles. À la racine, $\gamma_{\max}$ représente la performance du meilleur mouvement à partir de p. Il est ensuite possible de retrouver le meilleur mouvement en descendant l'arbre jusqu'à une feuille indiquant l'intervalle où déplacer p. À l'initialisation, la valeur de γ dans les feuilles correspond à la performance d'un mouvement déplaçant p vers l'intervalle représenté par la feuille et, dans les nœuds internes, cette valeur est à zéro. Ensuite, lorsqu'un mouvement $<\pi_i \rightarrow j>$ est appliqué, les arbres correspondants aux voisins de π_i sont mis à jour en modifiant la valeur de γ et $\gamma_{\max}$ dans les nœuds internes plutôt que dans chaque feuille entre i et j. Après une initialisation en $\mathcal{O}(m)$, l'exploration du voisinage peut être effectuée en $\mathcal{O}(n)$ puisque chaque arbre contient, à la racine, l'information nécessaire sur le meilleur mouvement. La mise à jour est faite en $\mathcal{O}(d_{max} \log d_{max})$, où d_{max} est le degré maximum d'un sommet dans l'exemplaire. La complexité d'une itération de descente est alors $\mathcal{O}(n + d_{max} \log d_{max})$ et utilise $\mathcal{O}(m)$ espace mémoire.

Implémentation arborescente MPS

Une implémentation différente utilisant aussi des arbres est présentée par Correal et Galinier [14]. Pour définir cette implémentation, nous décrivons un problème simplifié équivalant à trouver le meilleur mouvement à partir d'un élément dans l'implémentation quadratique. Pour une suite de valeurs S, le problème de somme partielle maximale (MPS – *Maximum Partial Sum*) consiste à obtenir j et $pmax = P_S(j)$ qui maximisent la valeur de $pmax$, où $P_S(j)$ est la somme des j premiers éléments de S. En assignant à S les valeurs de D utilisées dans l'implémentation quadratique pour un élément p, la solution au problème MPS indique que le meilleur mouvement à effectuer sur p est $<p \rightarrow j>$ avec une performance de $pmax$. Il est possible de résoudre ce problème avec une approche *diviser pour régner* en maintenant une structure arborescente. Pour deux suites L et R tel que leur concaténation donne S, on observe la relation de récurrence à l'équation 2.5.

$$\begin{aligned} S.somme &= L.somme + R.somme \\ S.pmax &= \max(L.pmax, L.somme + R.pmax) \end{aligned} \tag{2.5}$$

La figure 2.1 illustre un exemple de la structure de données arborescente MPS pour la séquence $S = (5, -2, 3, -4)$. On retrouve bien à la racine que la somme des éléments de S est 2 et que la valeur maximale de $pmax = P_S(j)$ est 6. Il est à noter que les valeurs de la séquence ne sont pas contenues dans la structure de données et sont affichées dans chaque nœud dans le but d'illustrer quels éléments sont représentés.

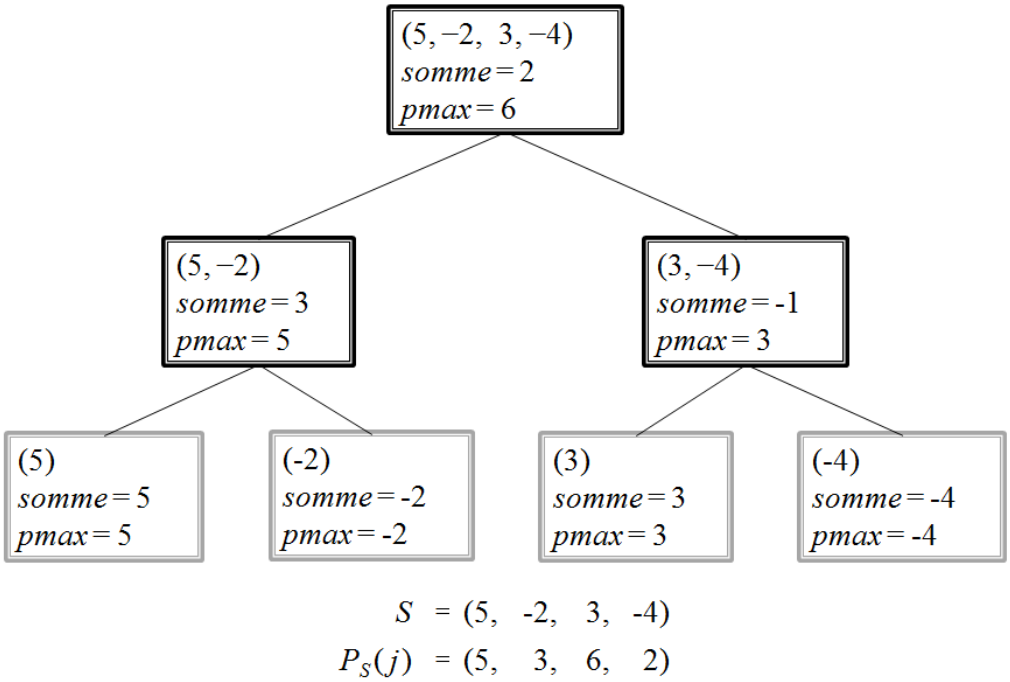


Figure 2.1 Exemple de la structure arborescente MPS pour la séquence $S = (5, -2, 3, -4)$

Ce principe est exploité pour implémenter l'exploration du voisinage d'insertion pour le LOP en créant deux arbres binaires pour chaque élément p de la configuration π. Les deux arbres représentent les mouvements de chaque côté de p. Pour chaque voisin q de p, une feuille avec la valeur $D[q][p]$ est incluse dans l'arbre approprié selon sa position dans π. Les nœuds internes maintiennent un attribut *somme* et *pmax* qui correspondent à la séquence obtenue en concaténant les valeurs dans toutes les feuilles sous-jacentes. La performance du meilleur mouvement à partir de p peut alors être obtenue en temps constant en accédant à la valeur de *pmax* à la racine des arbres associés à p. Il faut par contre maintenir la structure de données à jour. Suite à un mouvement $<p \rightarrow j>$, dans les arbres correspondants à un voisin de p, la feuille associée à p est déplacée en ajustant les valeurs contenues dans les nœuds jusqu'à la racine. De plus, les arbres qui représentent p doivent être reconstruits. L'initialisation d'un arbre se fait en $\mathcal{O}(d_{max})$, donc $\mathcal{O}(m)$ pour la structure complète. L'exploration du voisinage se fait en $\mathcal{O}(n)$ puisque le meilleur mouvement à partir d'une variable est disponible en temps

constant et la mise à jour se fait en $\mathcal{O}(d_{max} \log d_{max})$. La complexité d'une itération est donc $\mathcal{O}(n + d_{max} \log d_{max})$ et l'espace mémoire requis est $\mathcal{O}(m)$, comme pour l'implémentation arborescente originale. À la section 4.4, une extension de cette implémentation est présentée en détail.

Implémentation *dynasearch*

Bien que le voisinage *dynasearch* ait une taille exponentielle d'approximativement $\Theta(2,206^n)$ [13], Congram propose un algorithme de programmation dynamique capable de le parcourir en temps polynomial de $\mathcal{O}(n^2)$ [13] en mettant à profit la méthode séquentielle utilisée pour l'implémentation quadratique.

2.4 Métaheuristiques

Les métaheuristiques sont souvent utilisées pour résoudre des problèmes NP-difficiles lorsque les méthodes exactes ne sont plus praticables. De nombreuses études se sont penchées sur l'application d'une métaheuristique au problème LOP. Dans cette section, divers travaux et algorithmes notoires sont présentés. D'autres métaheuristiques ont aussi été proposées tel que la recherche locale avec relance [8, 31, 35], GRASP [6], colonie de fourmis [40], recuit simulé [35] et *Great Deluge Algorithm* [42].

2.4.1 Recherche tabou

La recherche tabou (TS – *Tabou Search*) est une métaheuristique introduite par Glover [18] en 1986. Cet algorithme utilise principalement la recherche locale comme dans une descente et y incorpore une mémoire afin d'effectuer un parcours plus intelligent et pouvoir sortir d'un optimum local. Le but de cette mémoire est en quelque sorte de se souvenir des solutions ou des régions de l'espace de recherche qui ont été précédemment visitées et ainsi de pouvoir empêcher d'effectuer un mouvement pour y retourner trop rapidement, ce qui annulerait l'effet de mouvements récemment effectués. Pour ce faire, une liste contient des attributs interdisant l'application de certains mouvements. L'algorithme de recherche locale doit alors trouver le meilleur mouvement admissible, même s'il dégrade la solution, et mettre à jour la liste tabou. Ainsi, lorsqu'un optimum local est atteint, un mouvement est tout de même appliqué pour s'en éloigner et la liste tabou empêche d'y retourner aux itérations suivantes. Les attributs sont conservés dans la liste pendant un certain nombre d'itérations fixé par un paramètre de l'algorithme. Ce paramètre a une influence importante sur la qualité des solutions trouvées par la recherche tabou, car une liste trop courte ne permettra pas à la

recherche de sortir d'un optimum local alors qu'une liste trop longue peut restreindre trop de mouvements, empêchant d'atteindre un optimum. De plus, un critère d'arrêt tel qu'un temps limite est nécessaire puisque l'algorithme ne s'arrête pas de lui-même.

La recherche tabou peut aussi utiliser des techniques d'intensification et de diversification. L'intensification a pour but de guider la recherche vers de bonnes solutions en retournant aux meilleures déjà trouvées alors que la diversification tente de visiter des régions de l'espace de recherche qui ont été peu visitées. Ces deux techniques, dont l'objectif est opposé, peuvent alors être utilisées à différents moments pour améliorer la qualité des solutions trouvées.

Un algorithme de recherche tabou est proposé par Laguna et al. [33] en 1999 surpassant les méthodes existantes à l'époque. L'algorithme alterne entre deux phases simples d'intensification et de diversification. De plus, chaque phase d'intensification se termine par une étape de recomposition de chemin (*path relinking*) [22] qui consiste à conserver un ensemble des meilleures solutions trouvées pour déterminer où déplacer, un à la fois, chacun des éléments de la configuration et périodiquement appliquer une recherche locale sur les configurations intermédiaires. Enfin, une diversification supplémentaire est effectuée lorsque le score de la meilleure solution ne s'est pas amélioré pendant plusieurs itérations. L'ensemble des solutions élites ainsi que les configurations évaluées pendant la dernière phase d'intensification sont utilisés pour calculer la position moyenne qu'occupe chacun des éléments. Puis, les éléments sont insérés à leur position complémentaire définie par $n - \alpha$, où α est la position moyenne d'un élément. La recherche locale dans cet algorithme utilise l'implémentation de base et la politique de recherche lors des descentes est FVI.

2.4.2 Recherche locale itérée

L'algorithme de descente est une façon simple et efficace de trouver un optimum local. Afin d'obtenir une solution de meilleure qualité, il est donc facile d'explorer d'autres optima locaux en effectuant plusieurs descentes à partir de solutions initiales différentes qui peuvent être générées par un algorithme de construction, aléatoirement, ou à partir d'une solution existante. L'algorithme de recherche locale itérée (ILS – *Iterated Local Search*) utilise ce principe pour continuer la recherche lorsqu'un optimum local est atteint. Dans ce cas, une perturbation aléatoire est appliquée à la solution courante et la recherche recommence à partir de la configuration obtenue. Lorsqu'une descente se termine, un critère d'acceptation détermine si la nouvelle solution est conservée ou si l'optimum local précédent est utilisé une fois de plus. Enfin, puisque cette technique ne s'arrête pas d'elle-même, un autre critère indique quand l'algorithme doit être terminé. Imposer un nombre de descentes fixe ou un temps maximal est un exemple commun de critère d'arrêt.

Un algorithme ILS a été implémenté pour le LOP par Congram [13] en 2000 en utilisant le voisinage *dynasearch*. La perturbation utilisée consiste en une série de mouvements d'insertion aléatoires et un nouvel optimum local est conservé seulement s'il est meilleur que la solution courante. Il s'agit de la première utilisation du voisinage *dynasearch* pour le LOP et cet algorithme donnait les meilleures performances en termes de qualité des solutions et de temps d'exécution par rapport à l'état de l'art quand il a été publié.

Un autre algorithme ILS est décrit par Schiavinotto et Stützle [43] en 2004. Celui-ci met à profit la politique de recherche FVI et l'implémentation quadratique. Plusieurs mouvements de type *interchange* sont appliqués pour les perturbations et le critère d'acceptation continue avec un nouvel optimum local en autorisant une légère dégradation.

Une étude plus poussée sur la recherche locale itérée est présentée par Valdez et Medina [44] en 2012 dans laquelle plusieurs variantes sont testées pour la construction de la solution initiale, l'opérateur de recherche, ainsi que la perturbation à effectuer. En particulier, la recherche locale itérée est améliorée par un bon équilibre entre l'intensification (recherche locale) et la diversification (perturbation).

Pour faire suite à l'introduction de l'implémentation arborescente [41] qu'ils ont proposée pour la recherche locale, une autre version de ILS utilisant cette implémentation et la politique BI est étudiée par Sakuraba et al. [42] en 2015. Les perturbations sont faites par plusieurs mouvements d'insertion aléatoires et le critère d'acceptation est de toujours conserver les nouvelles solutions. Ce critère d'acceptation semble motivé par le fait que pour réutiliser l'optimum local précédent lors d'un rejet, il faudrait garder en mémoire de grandes structures de données ou alors les reconstruire, ce qui constitue un effort de calcul trop pénalisant.

2.4.3 Recherche à voisinage variable

La recherche à voisinage variable (VNS – *Variable Neighborhood Search*) est une métaheuristique développée par Mladenović et Hansen [38] qui met à profit plusieurs voisinages différents lors de la recherche locale. Le principe de cette méthode est que, pour plusieurs problèmes, un optimum local dans un voisinage n'en est pas nécessairement un pour un autre voisinage et que les optima locaux de plusieurs voisinages sont relativement rapprochés. De plus, l'optimum global est aussi un optimum local dans tous les voisinages. L'algorithme standard débute avec une configuration initiale x et le premier voisinage. Puis, à chaque itération, un des voisins de x, noté x', est choisi aléatoirement. Une recherche locale est alors appliquée à x' résultant en un optimum local x''. Si cette solution est meilleure que x, l'algorithme recommence à partir de x'' avec le voisinage initial. Sinon, le voisinage suivant est sélectionné jusqu'à ce qu'ils aient tous été appliqués sans améliorer la configuration courante. On observe

donc trois étapes, une perturbation de la solution courante, une recherche locale, puis une mise à jour de la meilleure solution et du voisinage.

Différentes versions de VNS appliquées au LOP sont étudiées par Garcia et al. [16]. Le premier voisinage utilisé est swap et les suivants sont une combinaison de k mouvements d'insertion, pour $1 \leq k \leq k_{\max}$. Ainsi, un nombre arbitraire de voisinages peuvent être essayés. En plus de la version de base de l'algorithme, une heuristique est testée lors de l'étape de perturbation pour assurer une diversité des configurations de départ pour la recherche locale. Aussi, des versions hybrides sont évaluées dans lesquelles une recherche tabou effectue l'étape de recherche locale. L'algorithme incluant ces deux améliorations s'avère la version donnant les meilleurs résultats. L'absence de détails sur l'exploration des voisinages suggère que l'implémentation de base est utilisée.

2.4.4 Recherche dispersée

La recherche dispersée (SS – *Scatter Search*) est un type d'algorithme évolutionnaire. Un modèle de l'algorithme est présenté par Glover [19] dans lequel un ensemble de configurations relativement petit, nommé l'ensemble de référence, est sélectionné parmi une population diversifiée d'optima locaux. Ensuite, une nouvelle configuration est générée à partir de certains éléments de l'ensemble de référence, puis est améliorée par une recherche locale. L'ensemble de référence est ensuite mis à jour en considérant de remplacer une de ses solutions par la plus récente. La recherche se poursuit jusqu'à ce que toutes les combinaisons voulues de solutions dans l'ensemble de référence aient été testées. On distingue cinq étapes importantes dans cet algorithme : la génération d'une population diversifiée d'optima locaux, la méthode d'amélioration, la sélection et mise à jour de l'ensemble de référence, le choix d'un sous-ensemble de l'ensemble de référence et la combinaison de plusieurs configurations pour en créer une nouvelle. Il est à noter que l'ensemble de référence ne contient pas nécessairement uniquement les meilleures solutions puisqu'il est important de conserver une population diversifiée.

Un algorithme de SS est proposé pour le LOP par Campos et al. [6]. Dix générateurs de configurations sont comparés afin d'améliorer la qualité et la diversité de la population initiale. De plus, quatre collections de sous-ensembles de tailles différentes sont utilisées simultanément. La recherche locale est effectuée avec la politique *FVI* et l'implémentation de base. Les performances de cette méthode dépassent celles de la littérature lors de sa publication en 2001.

Un algorithme plus récent implémentant la recherche dispersée est étudié par Huacuja et al. [24]. Ils proposent de varier la version de la descente utilisée à différentes étapes de la recherche dispersée et affirment obtenir de meilleures solutions que les algorithmes de l'état de l'art. Par

contre, leur comparaison peut difficilement être analysée puisque leurs exécutions sont faites sur un ordinateur plus récent que celui des algorithmes de référence. Bien que le processeur ait une fréquence 18 fois plus élevée, la même limite de temps est appliquée.

2.4.5 Algorithme génétique et mémétique

L'algorithme génétique (GA – *Genetic Algorithm*) s'inspire de la théorie de l'évolution et de la sélection naturelle. Le principe est de simuler l'évolution d'un ensemble de configurations. Cet ensemble est appelé une population et chaque configuration est un individu. La population de départ est générée aléatoirement ou par une méthode de construction. Ensuite, une nouvelle génération d'individus est créée en effectuant des croisements ou des mutations, puis la population est mise à jour en éliminant les individus avec le score le plus faible pour ramener la population à la taille voulue. Cette étape est répétée jusqu'à ce qu'un critère d'arrêt soit satisfait. Un croisement consiste à créer un individu en combinant les caractéristiques de deux parents sélectionnés dans la population et une mutation modifie aléatoirement un individu. La manière d'effectuer un croisement et une mutation ainsi que la sélection des individus à traiter est propre à chaque algorithme. La taille de la population ainsi que le nombre de croisements et de mutations sont des paramètres à déterminer, souvent expérimentalement. Un tel algorithme est développé pour le LOP par Charon et Hudry [9], mais ne réussit pas à obtenir de bons résultats [35].

L'algorithme mémétique (MA – *Memetic Algorithm*) est basé sur le génétique et incorpore de la recherche locale à chaque génération. Dès la création de la population initiale, une recherche locale est appliquée sur chaque nouvel individu. Ainsi, la population est composée uniquement d'optima locaux.

Les deux types d'algorithme sont analysés par Huang et Lim [25] en 2003. Encore une fois, l'algorithme génétique donne des résultats de faible qualité. Par contre, l'algorithme mémétique est très efficace. Trois opérateurs de croisement et deux de mutation sont étudiés pour parfaire cette technique. La version donnant les meilleurs résultats n'utilise pas de mutation. La recherche locale est effectuée avec la politique *FI* et l'implémentation quadratique.

Un autre algorithme mémétique est présenté par Schiavinotto et Stützle [43] en 2004. Plusieurs versions de leur algorithme sont évaluées, testant entre autres quatre opérateurs de croisement. La plus efficace n'applique pas de mutation. L'opérateur de croisement choisi sélectionne une partie des éléments du premier parent et les réordonne selon leur position dans le second parent. Un élément qui distingue cet algorithme du précédent est l'utilisation de la politique *FVI* qui accélère énormément la recherche locale.

En 2014, Ye et al. [45] proposent un opérateur de croisement différent qui utilise plus de deux parents. Un algorithme mémétique avec ce croisement et une recherche locale suivant la politique *BI* est développé et inclut une nouvelle approche pour choisir les parents et mettre à jour la population afin de préserver la diversité des individus.

2.5 Comparaison des algorithmes

Une comparaison expérimentale de plusieurs algorithmes est effectuée par Martí et al. [36] en 2012. Des tests sont effectués avec les jeux de données LOLIB, SGB, MB, Random et XLOLIB présentés à la section 2.6 ainsi que quelques exemplaires spéciaux provenant de diverses publications. Les métaheuristiques testées sont entre autres la recherche tabou (TS) [33], algorithme mémétique (MA) [43], recherche à voisinage variable (VNS) [16], recuit simulé (SA) [36], recherche dispersée (SS) [6], recherche locale itérée (ILS) [43] et algorithme génétique (GA) [9].

Une première expérience traitant les jeux de données avec une solution optimale connue révèle que ces exemplaires sont assez faciles. En particulier, les métaheuristiques TS, ILS et MA trouvent la solution optimale pour presque tous les exemplaires. Les exemplaires plus difficiles, dont la solution optimale est inconnue, sont ensuite traités avec deux limites de temps. Après 10 secondes, l'algorithme MA est visiblement supérieur, suivi de près par ILS. Les méthodes TS, VNS et SS donnent aussi des résultats acceptables. En augmentant le temps d'exécution à 600 secondes, on observe des résultats semblables. Les cinq meilleurs algorithmes en terme de temps d'exécution et de qualité de solution sont, dans l'ordre, MA, ILS, TS, VNS, SS. Enfin, SA et GA obtiennent de mauvais résultats, même sur les exemplaires faciles.

Pour la plupart des métaheuristiques étudiées dans [36], l'algorithme utilisé est encore le plus récent. Par contre, il existe des versions plus récentes pour ILS et MA. Dans le cas de la recherche itérée, l'implémentation [44] surpasse la précédente, mais reste dominée par MA. L'implémentation subséquente de ILS dans [42] obtient des résultats compétitifs comparés à la version [43] et améliore les meilleures solutions pour le jeu de données créé par les auteurs, qui avait été traité par la descente uniquement auparavant. Il n'y a pas de comparaison entre [44] et [42]. Pour l'algorithme mémétique le plus récent, il est également difficile de le comparer avec la version de référence. Les résultats fournis indiquent seulement que l'algorithme est compétitif et que de meilleures solutions sont trouvées pour plusieurs exemplaires avec un temps d'exécution significativement plus élevé.

2.6 Jeux de données

De multiples exemplaires du problème ont été étudiés dans la littérature. Cette section décrit plusieurs des jeux de données mentionnés régulièrement.

LOLIB Les exemplaires de LOLIB (*Linear Ordering Library*) sont tirés de tableaux d'entrée-sortie réels pour divers pays européens. Il y a en tout 50 exemplaires de tailles variables entre 44 et 79 et la densité varie entre 28% et 95%, avec une majorité des exemplaires au-dessus de 80% de densité. Ce jeu de données a été traité en premier dans [23], puis repris par la suite sous le nom de LOLIB en ajustant les valeurs à des nombres entiers. Ce jeu de données est maintenant considéré comme relativement facile et les solutions optimales sont connues.

Stanford GraphBase – SGB Ce jeu de données est tiré de Stanford GraphBase [32] et est utilisé dans [33]. Il consiste en 25 exemplaires de taille 75 et de densité autour de 90% qui ont été créés de façon aléatoire à partir d'une distribution uniforme des valeurs $[0, 25000]$. Les optima sont connus pour tous les exemplaires.

Mitchell et Borchers – MB Ces exemplaires sont créés par Mitchell et Borchers [37] à partir d'une distribution uniforme $[0, 99]$ pour le triangle supérieur et $[0, 39]$ pour le triangle inférieur. Ensuite, une partie des valeurs est mise à zéro. La valeur de la solution optimale est connue pour les 30 exemplaires de tailles entre 100 et 250. La densité de ces exemplaires est au-dessus de 95%.

Random Ce jeu de données aléatoire est généré dans [6]. Le type Random A est construit à partir d'une distribution uniforme de $[0, 100]$ pour les tailles $n = \{100, 150, 200\}$. Des exemplaires additionnels de taille 500 ont été générés dans [36]. Il y a 25 exemplaires pour chaque taille et leur densité est d'environ 99%. Aucune solution optimale n'est trouvée pour ce jeu de données.

XLOLIB Le jeu de données XLOLIB est introduit dans [43] et est construit en échantillonnant aléatoirement les valeurs des exemplaires LOLIB. Il s'agit donc d'exemplaires aléatoires aux propriétés semblables aux jeux réels. Un exemplaire de taille 150 et un autre de 250 sont générés à partir de 49 exemplaires LOLIB. La densité varie entre 35% et 95%, dont la majorité au-dessus de 85%. Les solutions optimales ne sont pas connues pour ce jeu de données.

Sakuraba et Yagiura Ces exemplaires sont générés par Sakuraba et Yagiura [41] pour des tailles $n = \{500, 1000, 2000, 3000, 4000, 8000\}$ et des densités $d = \{1\%, 5\%, 10\%, 50\%, 100\%\}$. Pour chaque paire (n, d), cinq exemplaires sont créés en assignant une valeur aléatoire dans $[1, 99]$ avec une probabilité d à chaque élément de la matrice. Ce jeu de données contient donc 150 exemplaires de grandes tailles pour lesquelles la solution exacte est inconnue.

FSP Un jeu de données est généré spécifiquement pour la version non pondérée des problèmes *Feedback Vertex Set Problem* et FASP par Pardalos et al. [39] et est donc aussi valide pour le LOP. Ces exemplaires de tailles $n = \{50, 100, 500, 1000\}$ sont construits en choisissant aléatoirement m arcs dans un graphe pour 10 valeurs de m différentes pour chaque taille. Dans le cadre du LOP, on assigne aux arcs un poids de 1. La densité est très faible dans ces graphes, surtout pour les gros exemplaires, avec 11 cas sous 1% de densité sur 40 exemplaires. Les solutions exactes ne sont pas connues pour ces exemplaires.

CHAPITRE 3 MÉTHODOLOGIE

Le but de notre travail est essentiellement de développer et implémenter des opérateurs de recherche compétitifs pour le LOP en se basant sur ceux existant dans la littérature et ensuite d'analyser leurs performances. Ce chapitre décrit le cheminement du travail effectué pour atteindre cet objectif et introduit l'article au chapitre suivant, dans lequel les principaux résultats de la recherche sont présentés.

3.1 Démarche du travail de recherche

Au début du travail, nous avons commencé par développer une version préliminaire d'une structure de données arborescente permettant d'implémenter efficacement l'exploration du voisinage d'insertion LOP, ainsi qu'un opérateur de recherche l'utilisant. Cette implémentation est présentée à la conférence EvoCOP 2015 [14]. Par la suite, quelques modifications ont été apportées à cette implémentation afin de la rendre plus générique et applicable dans diverses situations. En particulier, elle est adaptée pour tirer profit de la faible densité de certains exemplaires. Avec ces améliorations, l'implémentation arborescente proposée et celle existant dans la littérature possèdent la même complexité théorique.

L'implémentation quadratique classique utilisée dans les heuristiques récentes appliquées au LOP est principalement conçue pour des exemplaires très denses. Nous avons donc implémenté, en plus de cette technique, une version améliorée lui permettant de s'adapter aux exemplaires peu denses.

Ensuite, nous avons développé des opérateurs de recherche avec chacune des implémentations proposées et plusieurs politiques de recherche. Ces opérateurs variés sont alors appliqués et comparés dans le cadre d'une descente à partir d'une solution aléatoire sur un jeu de données généré aléatoirement. De plus, nous avons analysé l'impact des caractéristiques d'un opérateur, c'est-à-dire l'implémentation et de la politique de recherche, sur le temps d'exécution de ces opérateurs en fonction de la taille et de la densité des exemplaires. Pour ce faire, un modèle simplifié est créé afin de représenter le temps d'exécution de l'algorithme de descente et ainsi isoler l'impact de chacune des caractéristiques sur une mesure plus spécifique. Cette analyse est aussi effectuée pour les descentes incorporées dans un algorithme mémétique afin d'évaluer les opérateurs dans le contexte d'une heuristique hybride.

Enfin, les résultats de la comparaison des opérateurs de recherche développés et de leur analyse détaillée sont utilisés pour déterminer le meilleur opérateur selon le jeu de données

traité.

Pour toute l'analyse, les comparaisons des divers opérateurs de recherche sont effectuées en se basant uniquement sur le temps d'exécution. Le score des solutions trouvées n'est pas considéré puisque les opérateurs testés utilisent tous le même voisinage et que nous supposons donc qu'ils peuvent théoriquement atteindre les mêmes optima locaux. De plus, nous évaluons principalement les opérateurs de recherche dans le cadre de descentes avec relances alors qu'une application plus intéressante de la recherche locale est de l'inclure dans une heuristique hybride. Dans ce cas, le type de l'heuristique hybride (par exemple l'algorithme mémétique ou la recherche locale itérée) et ses paramètres vont aussi influencer la qualité des solutions retournées par l'heuristique.

3.2 Présentation de l'article

L'article au chapitre 4 présente le travail principal effectué dans ce mémoire. D'abord, les opérateurs de recherche de la littérature qui sont considérés dans l'étude sont détaillés à la section 4.3.1 et leurs adaptations aux exemplaires peu denses sont expliquées à la section 4.3.2. Ensuite, la variante de l'implémentation arborescente est proposée à la section 4.4 ainsi que les différences de celle-ci avec l'implémentation semblable proposée auparavant dans la littérature. La section 4.5 décrit l'expérimentation effectuée. Puis, les résultats sont présentés et discutés à la section 4.6. Enfin, la section 4.7 résume les conclusions tirées dans l'article et les principales contributions.

CHAPITRE 4 ARTICLE 1 : Hill climbing heuristics for linear ordering : an empirical study

4.1 Introduction

The linear ordering problem (LOP) is a well-studied NP-hard combinatorial problem that has applications in various fields. Several exact algorithms have been proposed to the problem [23, 27, 37] along with a large number of different heuristics [6, 8, 13, 16, 25, 33, 43, 44, 45].

Most of the best-performing heuristics proposed to the LOP are hybrid heuristics that run repeatedly a hill climbing (HC) operator. It is notably the case for iterated local search (ILS) [13, 42, 43, 44], variable neighborhood search (VNS) [16], and memetic algorithms (MA) [25, 43, 45]. The efficiency of these hybrid algorithms depends therefore on the quality of the solutions returned by their HC operator and by its running time.

Starting from an initial configuration transmitted in argument, a HC operator performs a series of iterations. In each iteration, the algorithm selects and applies an improving move – i.e. a move that increases strictly the score of the current configuration. The algorithm stops when no more improving move exists, that is to say when the current configuration is locally optimum. The different moves available on each iteration are defined by the neighborhood. The criterion used to choose a move is named the policy. Another important feature of a HC operator is the implementation technique used to manipulate the neighborhood.

The most efficient neighborhood proposed to the LOP is the so-called insert neighborhood. The two most evolved implementation techniques devoted to this neighborhood are the so-called regular and tree implementations. Note that while the quality of the solutions returned by the HC operator depends mostly on the neighborhood, its running time also depends on its policy and on its implementation. In this paper, we first describe and analyze policies and implementation techniques proposed to the LOP while proposing several improvements. Then, we present an extensive experimental study in which different variants of the HC operator are compared by using a large set of problem instances of various size and density.

The remaining of this paper is organized as follows. In Section 4.2, we give a definition of the problem and present a review of literature; we also describe the neighborhoods proposed to the problem and analyze the implementation techniques designed for the insert neighborhood. We then detail the regular implementation and the tree implementation in Sections 4.3 and 4.4. Section 4.5 presents the experimental study. Section 4.6 details and analyzes the results. Finally, concluding remarks are proposed in the Section 4.7.

4.2 Background

4.2.1 Problem definition

A LOP instance is defined by an integer n and a n-by-n matrix C. A potential solution is any permutation π of $\{1..n\}$. The score of a permutation is defined according to Equation (4.1). The goal of the problem is to find a permutation of maximum score.

$$f(\pi) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} C_{\pi_i \pi_j} \tag{4.1}$$

The elements of $\{1..n\}$ will be named the variables of the problem. Finding a permutation consists therefore in finding the position of each variable $p = 1..n$. For every $p, q = 1..n$, we define D_{pq} according to Equation (4.2).

$$D_{pq} = C_{pq} - C_{qp}, \ \forall p, q = 1..n \tag{4.2}$$

Thus, Equation (4.3) holds, i.e. matrix D is antisymmetric.

$$D_{pq} = -D_{qp}, \ \forall p, q = 1..n \tag{4.3}$$

We can notice from Equation 4.1 that D_{pq} corresponds to the relative profit of placing p before q in the permutation, instead of placing q before p. In particular, if $C_{pq} = C_{qp}$ or, equivalently, if $D_{pq} = D_{qp} = 0$, it means that the relative position of p and q has no impact on the score of the permutation. In addition, when two variables p and q are such that $C_{pq} \neq C_{qp}$, we say that these two variables are neighbors. The density d of a problem instance will be defined as the ratio of neighboring variables: $d = 2 \times |\{(p, q) \in \{1..n\}^2 : p < q \text{ and } D_{pq} \neq 0\}|/(n \times (n-1))$. Also, we define the degree of a variable p as the number $deg(p) = |\{q \in \{1..p\} : D_{pq} \neq 0\}|$ of its neighbors and denote by $d_{max} = \max\{deg(p) : p = 1..n\}$ the maximum degree of a variable.

Finally, in our computer program, we represent the treated problem instance by using the input matrix C, the above-defined matrix D, and adjacency lists that indicate the neighbors of each variable.

4.2.2 Review of literature

The Linear Ordering Problem has applications in multiple fields, notably in economy as the triangulation of input-output tables [11]. Other common applications include the aggregation

of individual preferences [30], optimal weighted ancestry relationships [21], minimizing total weighted completion time in single-machine scheduling [3], ranking in sports tournaments [23] and one sided crossing minimization problem [26]. Another application arises in software development during integration tests in which various modules are tested together. In order to test a module, all of its dependencies also need to be integrated and already tested. Therefore, a stub needs to be created to break any cycle of dependencies. The problem of ordering modules to minimize the creation of stubs is called Class Integration Test Order (CITO) [4].

A graph problem equivalent to LOP is the maximum acyclic subgraph problem [23], in which the goal is to determine a subset of edges with maximum weight containing no directed cycle. An equivalent version is the Feedback Arc Set Problem (FASP), requiring to find a minimum set of edges that intersects all cycles.

A number of exact algorithms have been proposed to solve the LOP to optimality such as branch-and-bound [10, 15, 27], branch-and-cut [12, 23] and an interior point/cutting plane algorithm [37]. These algorithms are limited to relatively small instances.

Multiple heuristics have been proposed such as the CK method [8], using hill climbing, and various construction algorithms [1, 11]. The construction heuristic of Becker [2] is used in multiple other papers [5, 42, 44]. A more recent construction algorithm based on insertion [35] appears to give slightly better results.

A large set of metaheuristics have been applied to the LOP. The tabu search presented in [33] balances an intensification with path relinking and a specialized diversification phase. Multiple versions of iterated local search have been tested [13, 42, 43, 44]. A variable neighborhood search and an hybrid using tabu search are presented in [16]. Population-based metaheuristics are also proposed such as a scatter search [6], genetic algorithms [9, 25] and a few memetic algorithms investigating various crossover operators [25, 43] and a multi-parent crossover [45]. An extensive empirical comparison of the heuristics applied to the LOP is presented in [36]. According to this study, the most efficient heuristics are, in order, the memetic algorithm by Schiavinotto et Stützle [43], the ILS by the same authors [43], and the tabu search by Laguna et al. [33].

Local search plays an important role in most of the presented heuristics and its implementation has seen some improvements over the years. A sequential evaluation of the insert neighborhood is explained in [13], improving the complexity to $\mathcal{O}(n^2)$, from the basic implementation's $\mathcal{O}(n^3)$. Two different upgrades are proposed in [41] using additional data structures to explore the neighborhood with lists in $\mathcal{O}(nd_{max})$ or with trees in $\mathcal{O}(n+d_{max}\log d_{max})$. Finally, a static analysis of problem instances is proposed in [7] to restrict the search space by

detecting the positions where the elements of the configuration cannot generate local optima. However, the results reported in [7] indicate that the efficiency of this approach decreases for larger instances.

Several types of benchmarks have been used to test LOP heuristics. The usual LOP instances used in experiments are mostly dense, with sizes up to 500, and include real (LOLIB [23]), pseudo-real (XLOLIB [43]) or random data (Random A-B [6]). A set of larger random instances is generated in [41] with sizes from 500 to 8000 and various densities. Other very sparse instances are proposed for the FASP [39].

4.2.3 Neighborhoods proposed to the LOP

The two main neighborhoods proposed to the LOP are the interchange and the insert neighborhoods. The insert neighborhood is the one used in the best-performing LOP heuristics of the literature. It is shown in [25] that any local optimum with respect to the insert neighborhood is also a local optimum for the interchange neighborhood, while the reverse is false. As a matter of fact, HC operators that use the insert neighborhood reach on average better local optima than when using the interchange neighborhood, as observed in systematic experiments performed in [36]. In the following of our study, we only consider the insert neighborhood.

An insert move will be denoted by a couple $<p, j>$, where $1 \leq p \leq n$, $1 \leq j \leq n$, and $p \neq \pi_j$. Applying move $<\pi_i, j>$ to configuration π consists in moving variable $p = \pi_i$ from its current position i to a new position j, as expressed by Equation (4.4), where $\pi \oplus <\pi_i, j>$ denotes the configuration obtained by applying move $<\pi_i, j>$ to π. As each variable can take $n-1$ new positions, the size of the neighborhood equals $n \times (n-1)$.

$$\pi \oplus <\pi_i, j> = \begin{cases} (\pi_0, \ldots, \pi_{i-1}, \pi_{i+1}, \ldots, \pi_j, \pi_i, \pi_{j+1}, \ldots, \pi_n), & i < j \\ (\pi_0, \ldots, \pi_{j-1}, \pi_i, \pi_j, \ldots, \pi_{i-1}, \pi_{i+1}, \ldots, \pi_n), & i > j \end{cases} \tag{4.4}$$

Given a move $<p, j>$, we denote by $\delta(p, j)$ its score, i.e. its impact on the score of the current configuration, as expressed by Equation (4.5).

$$\delta(p, j) = f(\pi \oplus <p, j>) - f(\pi) \tag{4.5}$$

A positive (negative) value of $\delta(p, j)$ indicates therefore that applying move $<p, j>$ would increase (decrease) the score of the configuration. A move whose score is positive will be named an improving move. In addition, for a variable p, we denote by $\delta_\star(p)$ the score of the

best move applicable to p; if variable p is such that $\delta_\star(p) > 0$, we name it an improvable variable.

4.2.4 Policies

Two common generic policies used in HC operators are the *first improvement* (FI) and *best improvement* (BI) policies. Another policy which is different from FI and BI is exploited in [6, 16, 43]: we name it the *first variable improvement* policy (FVI).

In a BI iteration, the best possible move (i.e., the one having the maximum score) is applied. In a FI iteration, the moves are taken according to a particular order, and the first improving move is applied. A FVI iteration can be described as follows: take the variables according to a particular order; determine for each variable the best possible move applicable to this variable; if this move is an improving move, perform this move.

A natural and common way to perform a BI iteration is to evaluate each move one after the other. This technique can be very time consuming. The idea behind the FI and the FVI policies is of course to explore only a reduced portion of the neighborhood, so as to make an iteration less costly. Besides, we can observe that, with the FVI policy, the chosen variable is always moved to a new optimal position, while it is not the case with the FI policy.

Note that the order in which the variables are evaluated with the FVI policy can have an important impact on the efficiency of the policy. When using a fixed ordering of the variables as a reference, the process of evaluating them sequentially, starting from the first one at each iteration, is considerably slower than cycling through them, evaluating first the variable that follows the last one tested in the previous iteration.

4.2.5 Implementation techniques proposed to the LOP

What we name a neighborhood implementation technique is a set of procedures (along with eventual data structures) whose purpose is to perform low-level operations, notably evaluating the score of a given move, finding the best move applicable to a given variable, etc. The implementation technique may have a major impact on the speed of a local search algorithm, as it is common that a technically advanced implementation technique proves to be several orders of magnitude faster than a more basic one. We have identified in the literature four different implementation techniques devoted to the LOP insert neighborhood. We name these implementations the basic [33], the regular [13, 43], the list [41] and the tree [41] implementations. In this section, we present the main principles of these implementation techniques. More details about the regular and the tree implementations will be given in the

next sections.

In the **basic implementation** a given move is evaluated in $\mathcal{O}(n)$ thanks to Equation (4.6). This makes it possible to explore the whole neighborhood in $\mathcal{O}(n^3)$.

$$\delta(\pi_i, j) = \begin{cases} \sum_{k=i+1}^{j} D[\pi_k][\pi_i], & i < j \\ \sum_{k=j}^{i-1} D[\pi_i][\pi_k], & i > j \end{cases} \tag{4.6}$$

The idea behind the **regular implementation** is that the best move applicable to a given variable can be found in only $\mathcal{O}(n)$, thanks to recurrence Equation (4.7). As a result the whole neighborhood can be explored in $\mathcal{O}(n^2)$ by testing each of the n variables.

$$\begin{aligned} \delta(\pi_i, i) &= 0 \\ \delta(\pi_i, j) &= \delta(\pi_i, j-1) + D[\pi_j][\pi_i] \end{aligned} \tag{4.7}$$

In other words, with the regular implementation, evaluating each new move is done in constant time (instead of $\mathcal{O}(n)$ in the basic implementation); however, this imposes to scan the neighborhood in a particular order, as $\delta(p, j)$ is computed by using $\delta(p, j-1)$ or $\delta(p, j+1)$. More details about this implementation are given in Section 4.3.

From equations (4.6) and (4.7), we deduce that $\delta(p, j) = \delta(p, 1) + \sum_{k=1}^{j} D[\pi_k][p]$. Therefore, finding the best position j of variable p boils down to maximizing $\sum_{k=1}^{j} D[\pi_k][p]$, as $\delta(p, 1)$ is constant during the search. We note S^p the ordered sequence corresponding to $\{D[\pi_k][p] : k = 1..n\}$.

The principle of the **list implementation** is to store in a list the sequence S^p, for each variable $p = 1..n$. The interesting point is that only non-null elements need to be stored in the list. It is then possible to find the best move applicable to a given variable by scanning the list, in $\mathcal{O}(d_{max})$. However, after a move $m = <p_m, j_m>$ has been performed, the lists associated to the neighbors of p_m have to be updated (in $\mathcal{O}(d_{max}^2)$) so as to reorder their elements with respect to the new configuration – see more details in [41].

The principle of the **tree implementation** is to represent the values of S^p in the leaves of a tree structure. Using a tree structure of course makes it possible to update a list efficiently in $\mathcal{O}(\log d_{max})$. Most interestingly, the technique proposed in [41] makes it possible to find the best position of p without scanning the leaves by updating adequately additional fields stored in the internal nodes of the tree – see [41].

We can notice that, in addition to the current configuration, the list and the tree implementations need extra data structures (the lists and trees associated to the variables), and

that these data structures must reflect at any time the order of the variables in the current configuration; thus, these data structures must be initialized before the first iteration, and then updated at each iteration, after performing a move. On the other hand, we notice that the basic and the regular implementation techniques do not use specific data structures.

In summary, each implementation technique possesses procedures that perform the following operations : the initialization of the data structures; the discovery of the best move associated to a given variable; the updating of the data structures. The complexity of these operations under the different implementations is given in Table 4.1, along with the complexity of a HC iteration that explores the whole neighborhood (such as BI) and the space complexity of the extra data structures.

4.3 The regular and regular+ implementations

In this section, we detail how we have implemented the regular neighborhood implementation technique. We also detail a variant of the regular implementation technique, named regular+, that is intended to better deal with sparse problem instances. For each implementation technique, we describe the following low-level procedures:

— Procedure *findBestPosition*(p) returns the best position for the transmitted variable p along with the score of the corresponding move;
— Procedures *HC_FVI*() and *HC_BI*() perform a HC according to the FVI policy and the BI policy, respectively.

The pseudocode of the *findBestPosition*(p) procedure is given below. The score of each move $<p, j>$ (denoted by *delta*) is computed according to Equation (4.7): first for the positions $j > i$ (in the first loop), and then for the positions $j < i$ (in the second loop), where i is the rank of p, i.e. $\pi_i = p$. The procedure returns in $\mathcal{O}(n)$ the best new position *jbest* of variable p and the score of move $<p, jbest>$.

	init	**best move**	**update**	**BI iteration**	**size**
basic	-	$\mathcal{O}(n^2)$	-	$\mathcal{O}(n^3)$	-
regular	-	$\mathcal{O}(n)$	-	$\mathcal{O}(n^2)$	-
list	$\mathcal{O}(nd_{max})$	$\mathcal{O}(d_{max})$	$\mathcal{O}(d_{max}^2)$	$\mathcal{O}(nd_{max})$	$\mathcal{O}(nd_{max})$
tree	$\mathcal{O}(nd_{max})$	$\mathcal{O}(1)$	$\mathcal{O}(d_{max} \log d_{max})$	$\mathcal{O}(n + d_{max} \log d_{max})$	$\mathcal{O}(nd_{max})$

Table 4.1 Implementations: space and time complexity of the main operations

Procedure *findBestPosition*(p)
    $i := \pi.rank(p)$
    $deltaMax := -\infty$; $delta := 0$
    for $j := (i+1)..n$ **do**
        if version 1 **then** $delta := delta + D[\pi_j][p]$
        else if version 2 **then** $delta := delta - D[p][\pi_j]$
        Update *deltaMax* and *jbest*
    $delta := 0$
    for $j := (i-1)..1$ **do**
        $delta := delta + D[p][\pi_j]$
        Update *deltaMax* and *jbest*
    return (*delta*, *jbest*)

Version 1 of this procedure is more straightforward. Version 2 uses the antisymmetry property of D to apply a seemingly innocent modification and produces the same result – see Equation (4.3). The idea behind this is that it is faster to iterate the values of a matrix in a row than in a column. An experimental comparison that evaluates the impact of this modification will be presented in Section 4.6.5.

Besides, note that we have not implemented the first improvement (FI) policy as it does not appear promising. In particular, applying a FI policy in which the different moves are taken in a random order would require each move to be evaluated inefficiently in $\mathcal{O}(n)$.

4.3.1 Performing a hill climbing iteration with the regular implementation

We have implemented a procedure named *findMove_BI*() that returns a move according to the BI policy. This procedure simply tests each variable $p = 1..n$ one after the other using *findBestPosition*(p) to return the best move in $\mathcal{O}(n^2)$. We have also implemented a procedure named *findMove_FVI*() that returns a move according to the FVI policy. This procedure tests iteratively the variables according to a cyclic order until it finds an improvable variable, with a worst-case complexity of $\mathcal{O}(n^2)$. These two procedures are called repeatedly until no more improving move is found in *HC_BI*() and *HC_FVI*(), respectively.

We can notice that, when applying the *findMove_BI*() or the *findMove_FVI*() procedure, the number of elementary operations is proportional to $n-1$ multiplied by the number of tested variables. All n variables are tested when the BI policy is applied. With the FVI policy, the number of tested variables may vary between 1 and n from one iteration to the

other. Thus, the gain of efficiency of FVI over BI is related to the number of tested variables in FVI. Experiments related to this point will be reported in Section 4.6.3.

4.3.2 Performing a hill climbing iteration with the regular+ implementation

We can observe that the following property holds: the score $\delta_\star(p)$ of the best move applicable to a variable p will not change until a move is applied to p or to any of its neighbors. The regular+ implementation uses the same *findBestPosition*(p) procedure as the regular implementation. On the other hand, it exploits the above property in order to avoid testing some variables – both in the BI and in the FVI policy. The pseudocode of the *HC_FVI*() procedure is given below.

```
Procedure HC_FVI()
    Insert 1..n into Cand in a random order
    while Cand ≠ ∅ do
        p := Cand.dequeue()
        (delta, j) = findBestPosition(p)
        if delta > 0 then
            π := π.applyMove(p, j)
            for each q ∈ 𝒩(p), taken in a random order do
                if q ∉ Cand then
                    Cand.enqueue(q)
    return π
```

Note that the variables are evaluated according to a cyclic and randomized order – see Section 4.2.4. In the above procedure, *Cand* represents the set of candidate variables; it is implemented by a queue.

In the regular+ implementation of the BI policy, we use an array named *bestDelta*[ ] in order to store the values of $\delta_\star()$. When a move is applied to a variable p, *bestDelta*[p] is set to 0 and *bestDelta*[q] is set to a null value for every neighbor q of p, indicating that $\delta_\star(q)$ is unknown. The values contained in array *bestDelta* are exploited in order to avoid recalculating $\delta_\star(p)$ when *bestDelta*[p] $\neq$ *null*, since the value contained in *bestDelta*[p] is still valid. Finally, a variable p can be excluded from the set of candidate variables *Cand* if *bestDelta*[p] $\neq$ *null* and *bestDelta*[p] ≤ 0.

We can observe that, when the density equals 100%, the regular and regular+ implementations of a HC iteration (whether the FVI or the BI policy is used) test the same variables.

Therefore, a speedup is only expected for sparse instances. Also, the complexity of a HC iteration for both policy is $\mathcal{O}(n^2)$. An experimental comparison between the regular and regular+ implementations will be presented in Section 4.6.3.

4.4 The tree implementation

In this section, we detail how we have implemented the tree neighborhood implementation technique. The difference between our implementation and the one by the original authors [41] will be commented in Section 4.4.2. In the following of this section, we present successively what we name the maximum partial sum (MPS) abstract data type, its implementation, and how it is used in order to explore the LOP insert neighborhood in $\mathcal{O}(n + d_{max} \log d_{max})$.

4.4.1 The MPS abstract data type

Let S be a finite sequence of distinct elements of a set Ω, each element p in the sequence being assigned a particular value $v(p) \in \mathbb{R}$. For any $p \in S$, we define $\Sigma(p)$ as the sum of the values of the elements that precede p in S, including p itself:

$$\Sigma(p) = \sum\{v(q) : q \in S, q \preceq p\} \tag{4.8}$$

This function will be named the *partial sum function* associated to S. In addition, we define $Max\Sigma$ as the maximum value of the partial sum (see Equation 4.9), and $argMax\Sigma$ as the element $p \in S$ such that $\Sigma(p) = Max\Sigma$.

$$Max\Sigma = \max\{\Sigma(p), p \in S\} \tag{4.9}$$

We assume that S is first initialized, and that it thereafter undergoes a series of insertion and removal operations. Our goal is to determine the values of $Max\Sigma$ and $argMax\Sigma$ after each new update operation. Therefore, the MPS must support the following operations.

— Procedure $init(L, V)$ initializes the value of S with the elements of list L whose values are contained in list V.
— Procedure $insert(p, v, geq)$ inserts in S a new element p, whose value is v; geq is a Boolean function that indicates, for any two elements p and q of S whether p precedes q in the sequence or not.
— Procedure $remove(p)$ removes element p from S.

Finally, the MPS data type also implements three other procedures named $getMax()$, $getArgMax()$

and *next*(p); these methods return $Max\Sigma$, $argMax\Sigma$ and the element following p in S, respectively.

In the following example, we represent S and Σ ; for each element, we indicate first its value and then its identifier in brackets.

$$S = (3_{[6]}, -1_{[5]}, -3_{[3]}, 5_{[8]}, -2_{[7]}, 1_{[2]})$$
$$\Sigma = (3_{[6]}, 2_{[5]}, -1_{[3]}, 4_{[8]}, 2_{[7]}, 3_{[2]})$$

We notice that $Max\Sigma = 4$ and $argMax\Sigma = 8$. Assuming that operation remove(7) is applied to S, we now have that:

$$S = (3_{[6]}, -1_{[5]}, -3_{[3]}, 5_{[8]}, 1_{[2]})$$
$$\Sigma = (3_{[6]}, 2_{[5]}, -1_{[3]}, 4_{[8]}, 5_{[2]})$$

Also, $Max\Sigma = 5$ and $argMax\Sigma = 2$.

4.4.2 The MPS data structure

The MPS data structure is a tree data structure used to implement the above described MPS data type. Before detailing this data structure, let us first present an important property. Considering three sequences S, L and R such that $S = L \circ R$ (i.e., S is the concatenation of L and R) and $|L|, |R| \geq 1$, we claim that Equations (4.10) hold, where the attributes *first*, *sum* and *pmax* correspond respectively to the first element, the sum of all elements and the MPS of a sequence :

$$S.first = L.first \tag{4.10a}$$
$$S.sum = L.sum + R.sum \tag{4.10b}$$
$$S.pmax = \max(L.pmax, L.sum + R.pmax) \tag{4.10c}$$

The MPS data structure is organized as a balanced binary tree, implemented as an AVL tree, in which each node N corresponds to a sub-sequence $\mathsf{N}.seq$ of S. This tree corresponds to a recursive subdivision of S into sub-sequences: the root corresponds to S; each internal node has two sons $\mathsf{N.left}$ and $\mathsf{N.right}$ such that $\mathsf{N}.seq = \mathsf{N.left}.seq \circ \mathsf{N.right}.seq$ and each leaf corresponds to a sequence of size 1, i.e. to a single element.

In each node N, we memorize the following information: $\mathsf{N.first}$, $\mathsf{N.sum}$, and $\mathsf{N.pmax}$. A

node also has three links towards its father and its two sons. Note that the sequence itself ($\mathsf{N}.seq$) is not stored in the node. In a leaf representing an element p, `first` indicates the identifier p and both `sum` and `pmax` store the value $v(p)$ associated to this element.

Let us now describe the implementation of the procedures. The *init*() procedure builds a balanced tree that corresponds to the transmitted sequence. This is done by recursively creating the nodes and initializing their fields according to Equations (4.10). A procedure named *locate*(p) is used to locate the leaf who represents p by starting from the root and recursively selecting a child of the current node N with a comparison of p and $\mathsf{N}.\mathsf{right}.\mathtt{first}$. If a leaf with identifier p exists, the procedure finds it; otherwise, it returns the only leaf N such that $\mathsf{N}.\mathtt{first} \preceq p \preceq next(\mathsf{N}.\mathtt{first})$. An implementation optimization of this procedure uses an array of size n containing, for every $p = 1..n$, a link to the corresponding leaf if it exists, or a null value otherwise. It is then possible to skip the recursive process when the corresponding leaf is contained in the tree. The *insert*() procedure has three phases: first, the position of the new node is located; second, the new node is created, initialized and attached in the tree; third, the ancestors of the inserted node are updated according to Equations (4.10). The *remove*(p) procedure locates and removes the leaf whose identifier equals p, and updates the ancestors of the removed node. Note that procedures *init*(), *insert*() and *remove*() ensure that the structure of the tree remains balanced. The *getMax*() procedure simply returns $Max\Sigma = \mathsf{root}.\mathsf{pmax}$. The *getArgMax*() procedure starts from the root; then, it iteratively steps to a son of the current node N until it reaches a leaf by selecting the left child if $\mathsf{N}.\mathsf{left}.\mathtt{pmax} \geq \mathsf{N}.\mathsf{left}.\mathtt{sum} + \mathsf{N}.\mathsf{right}.\mathtt{pmax}$, or the right child otherwise.

As the data structure is maintained a balanced tree, it has $|S|$ leaves, $|S| - 1$ internal nodes, and its height is $\mathcal{O}(\log |S|)$. As a result, the *getMax*() procedure is $\mathcal{O}(1)$; the *locate*(), *insert*(), *remove*(), and *getArgMax*() procedures are $\mathcal{O}(\log |S|)$; finally the *init*() procedure is $\mathcal{O}(|S|)$.

Finally, let us comment on the differences between the original tree implementation proposed by Sakuraba et Yagiura [41] and ours. In the original implementation, the information contained in each node has no specific meaning and it is only used to make it possible to maintain the value of $Max\Sigma$ at the root of the tree. Just the opposite, in our implementation, the information we store in each node has a specific meaning as it corresponds to the sum and the $Max\Sigma$ of the sequence represented by the node. Also, the updating algorithm we use corresponds to a well-defined mathematical property defined in Equations (4.10). These differences make the updating algorithm and the whole technique easier to understand than in the original version.

4.4.3 Modeling the LOP neighborhood using the MPS

In order to implement the LOP insert neighborhood we use, in addition to the current configuration π, the following two main data structures: an array of n integers named *current*[ ] and an array of n MPS data structures named *tree*[ ].

At any time during the search, the data structures satisfy the two following properties. First, the value of *current*[p] equals $\delta(p, 1)$. Second, for each $q \in \mathcal{N}(p)$, we have in *tree*[p] an element whose identifier is q and whose value is $D[q][p]$. We also have in *tree*[p] an additional dummy element whose identifier and value both equal zero. Element 0 is the first in the sequence, while the other elements are ordered according to the rank they have in π.

As a result of the above properties, we have that *current*[p] corresponds to the variation of $f(\pi)$ that would occur assuming that p is removed from π. Also, *tree*[p].*getMax*() corresponds to the increase of $f(\pi)$ that would occur assuming that p is then inserted into π at a position j between the elements q and *tree*[p].*next*(q), where q = *tree*[p].*getArgMax*(). Therefore, the movement $<p, j>$ moves p to an optimal position and $\delta(p, j)$ = *tree*[p].*getMax*() + *current*[p].

Before the first iteration, the data structures are initialized with respect to the initial configuration according to the above properties. Then, after each move $<p, j>$, they are updated according to procedure *updateAfterMove*(p, j) whose pseudocode is given below. Note that, on the last line of the procedure, the value of *current*[p] is updated in constant time using *tree*[p].*getMax*(), which gives the contribution of p to the score in its best position. It is therefore only possible to use this value when p is moved to an optimal position, which is always the case during HC with the FVI or BI policies. In a different situation, the new value is calculated in linear time without changing the complexity of the procedure.

Procedure *updateAfterMove*(p, j)
    for every $q \in \mathcal{N}(p)$ **do**
        tree[q].*remove*(p)
        tree[q].*insert*($p, D[p][q], \pi.rank$)
        if $\pi.rank(p) < \pi.rank(q) \leq j$ **then**
            current[q] := *current*[q] − $D[p][q]$
        else if $\pi.rank(p) > \pi.rank(q) \geq j$ **then**
            current[q] := *current*[q] + $D[p][q]$
    current[p] := −*tree*[p].*getMax*()

We give below the pseudocode of the procedure *findMove_BI*() that returns a move according to the best improvement policy. Note that we have also developed a similar procedure

findMove_FVI() that returns a move according to the FVI policy. This procedure is similar to *findMove_BI*() except that it stops as soon as an improving move is discovered and tests the variables $1..n$ in a cyclic pattern.

Procedure *findMove_BI*()
    $deltaMax := -\infty$
    for $p := i..n$ **do**
        $delta := tree[p].getMax() + current[p]$
        Update *pBest*, *deltaMax*
    $jBest := getBestPosition(pBest)$
    return $(pBest, jBest)$

The pseudocode of the *getBestPosition*() procedure is given below.

Procedure *getBestPosition*(p)
    $i_p := \pi.rank(p)$
    $a := tree[p].getArgMax()$
    $i_a := \pi.rank(a)$
    if $i_a \geq i_p$ **then**
        $iBest := i_a$
    else
        $iBest := i_a + 1$
    return *iBest*

Note that the *updateAfterMove*() procedure is $\mathcal{O}(d_{max} \log d_{max})$ and both *findMove_BI*() and *findMove_FVI*() are $\mathcal{O}(n)$. The complexity of a HC iteration is therefore $\mathcal{O}(n+d_{max} \log d_{max})$. Also, the use of n trees takes up $\mathcal{O}(nd_{max})$ space in memory.

4.5 The experimental study

In this section, we present extensive experiments in which the running times of different variants of the HC operator are compared on a set of randomly generated problem instances. The benchmarks, the setting of the experiments, and the methodology applied to analyze the results are detailed in the remaining of this section.

4.5.1 Problem instances

We have generated problem instances of various size n and density d : $n = 63, 125, 250, \ldots, 8000$; $d = 1, 5, 10, 25, 50, 100\%$. Each combination of (n, d) is used to create 5 problem instances. There are therefore $8 \times 6 \times 5 = 240$ problem instances. We have built these LOP instances according to a random model similar to the one described in [41] and to the random instances of type A mentioned in [36].

The construction of a problem instance is as follows. The instance is represented as a directed graph, starting with a set of n disconnected vertices. The number of edges to create to reach the desired density d is $N_E = d \times n \times (n-1)/2$, rounded up to the nearest integer. The edges are then created by selecting N_E pairs of vertices uniformly and assigning to their extremities two distinct values in the range $[0, 99]$. Note that the small very sparse instances are very easy. Indeed, the combinations $(n, d) = (63, 1\%)$ only contains 20 edges, resulting in a disconnected graph. Therefore, these problem instances are ignored in detailed analysis where they are outliers.

4.5.2 Setting of the experiments

In our experiments, we use the six variants of the HC operator defined by the two considered policies (FVI and BI) and the three implementations (reg, reg+, and tree). Each variant is run on each problem instance in two different contexts: as a stand-alone HC with a randomly generated initial configuration (HCR), and embedded in a memetic algorithm (MA). The MA uses the parameters recommended in [43] and stops after the average score of the population stays the same for 30 consecutive generations. The HCR algorithm is run between 10 and 250 times, and the MA between 1 and 25 times, depending on the size of the instance. In addition to the six variants we implemented, we also run on our computer the HC operator provided by the authors of [43], which we refer to as FVI-reg-04 [1].

For a given HC run, we record, in addition to the input variables (the size n and the density d of the problem instance; the neighborhood implementation technique *impl*; the policy *pol*; and the context *ctx*), the following output variables: the score of the solution returned by HC (*score*); the running time to reach a local optimum (*cpuHC*); the number of iterations (*nbIt*) and the number of elementary operations (*nbOp*) performed during the run. The values of these output variables are averaged for the runs having a same value of n, d, *pol*, *impl*, and *ctx*. Table 4.2 enumerates these variables as well as some variables used for analysis.

1. The authors thank Tommaso Schiavinotto and Thomas Stützle for providing the source code of their MA algorithm.

All presented experiments have been performed on a machine running an Intel Core i7-3770 cpu at 3.4GHz with 16GB of RAM memory.

4.5.3 Methodology to analyze the results

In order to better understand the results of our experiments, we will conduct an in-depth analysis. The principle of this analysis is to count the number of elementary operations performed by the different variants of HC operator and to time these operations. For simplicity, we only consider the critical type of operation in which the largest part of the running time is spent. We define a (critical) elementary operation for the regular and regular+ implementations as one pass through a loop in procedure *findBestPosition*(p) described in Section 4.3. For the tree implementation, an elementary operation is defined as either updating the values in a node or selecting the next node in the *locate*(p) procedure. While both operations are different, we consider them equivalent for simplicity. Note that, to concentrate our analysis on the iterative part of the local search, no elementary operations are counted during the initialization of the trees.

It is important to note that the running time of an elementary operation is in theory constant. However, in practice, it may depend on different factors, in particular the size of the treated problem instance. For the sake of analysis, we will decompose the running time of an elementary operation as the product of a reference time (measured on a small problem instance) and the so-called slowing factor. In addition to the main experiment, a separate experiment was set up to isolate the execution of the elementary operations in all implementations, so as to evaluate more precisely the average running time of an elementary operation (*cpuOp*), for each triplet (n, d, *impl*). The reference time to perform an elementary operation *cpuOpRef* was fixed to the value of *cpuOp* that corresponds to $(n, d) = (63, 100)$ for the regular and regular+ implementations and measured on a single tree of size 50 for the tree implementation. Finally, the slowing factor (*slowF*) equals *cpuOp*/*cpuOpRef*.

4.6 Computational results

In this section, we present the results of the experiments described in Section 4.5. We first perform a comparison and a detailed analysis of the HC operator variants when initialized with a random configuration. Then, we observe how the different operators perform in the context of a memetic algorithm. Finally, we compare our results with those presented in [41]. The execution times per HC for every operator variant are detailed in Appendix A.

	variable	description
input	n	size of the problem instance; $n \in \{63, 125, 250, ..., 8000\}$
	d	density of the problem instance; $d \in \{1, 5, 10, 25, 50, 100\%\}$
	ctx	context; $ctx \in \{HCR, MA\}$
	pol	policy; $pol \in \{FVI, BI\}$
	$impl$	neighborhood implementation technique; $impl \in \{reg, reg+, tree\}$
output	$score$	score of the output solution
	$cpuHC$	running time per HC
	$nbIt$	number of iterations per HC
	$nbOp$	number of elementary operations per HC
analysis	$cpuOp$	real running time per elementary operation
	$cpuOpRef$	reference running time per elementary operation
	$slowF$	slowing factor

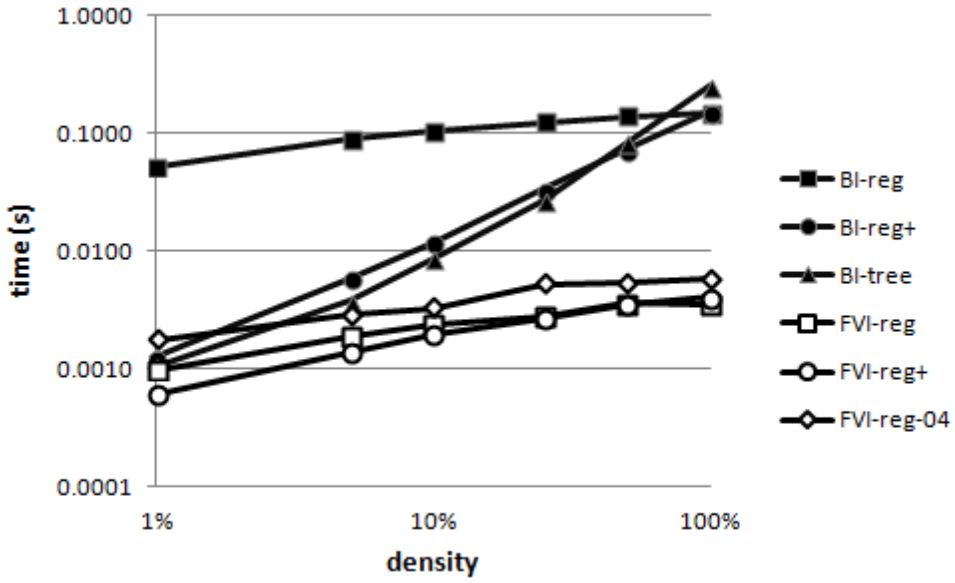
Table 4.2 Variables involved in our experiments

4.6.1 General comparison

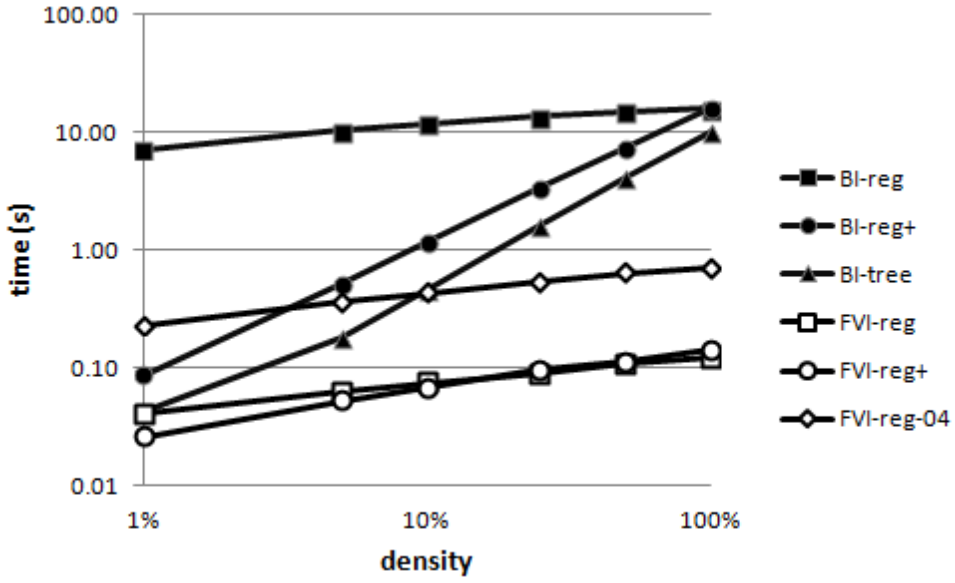
Figures 4.1a and 4.1b present the general results of the experiments by displaying the average execution time of a HC using the different variants in the context of HCR for $n = 500$ (Figure 4.1a) and $n = 2000$ (Figure 4.1b). For clarity, FVI-tree is not shown as it is very similar to BI-tree and constantly slower. We first observe that the execution time of the various HC operators differ for a same instance by up to two orders of magnitude. From the two figures, we can make the following remarks. The FVI policy is clearly more efficient than BI, especially on dense instances. FVI-reg+ and FVI-reg are the fastest operators for sparse and dense instances respectively. Also, FVI-reg and FVI-reg-04 are similar on small instances but the FVI-reg operator gets much faster as the size grows. We notice that BI-reg is particularly slow and is increasingly dominated by BI-reg+ as the density diminishes. BI-tree is much more efficient on sparser instances than it is on dense ones. Finally, the tree implementation is the best one for the BI policy on large instances.

4.6.2 Number of iterations per hill climbing

First, note that the number of iterations of a HC operator depends only on the policy and the instance, but not on the implementation. Figure 4.2 presents the number of iterations taken by FVI and BI during a HC for instances of size $n = \{500, 1000, 2000\}$. We observe that FVI always performs more iterations to reach a local optimum. Also, as the size or the density grows, both policies perform more iterations. The size of the instance in particular has a significant impact as the number of iterations increases faster than n does. However,



(a) $n = 500$



(b) $n = 2000$

Figure 4.1 Average execution time of a HC in the context of HCR depending on density, for problem instances of size $n = 500$ (Figure a) and $n = 2000$ (Figure b)

the ratio between the two policies increases only very slowly for larger and denser instances, ranging from 1.3 to 1.9 in our data set.

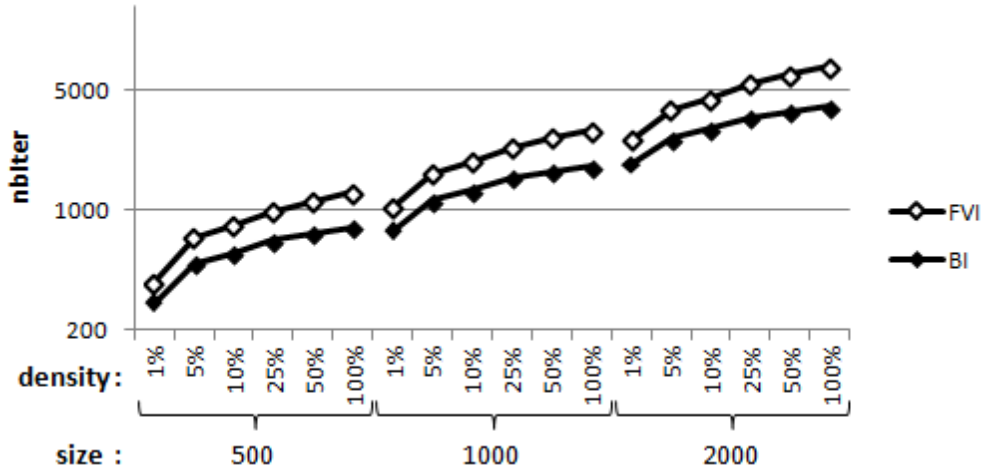


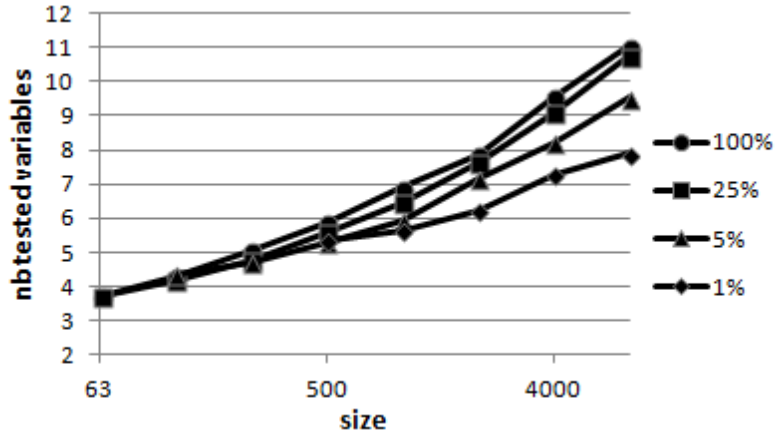
Figure 4.2 Average number of iterations per HC for the FVI and BI policies, for problem instances of size 500, 1000, and 2000

The different number of iterations for both policies can be explained because, during each iteration, BI needs to test n variables while FVI stops at the first improving one. This makes an FVI iteration faster, but also yields a lesser movement on the configuration. These smaller increments increase the number of iterations required to obtain a local optimum.

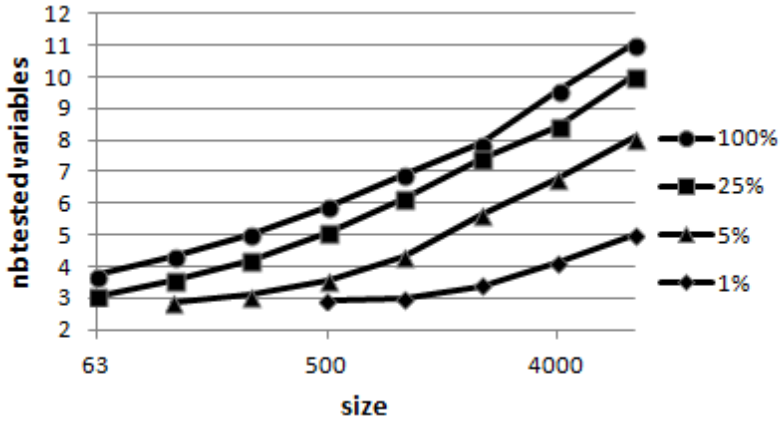
4.6.3 Elementary operations in the regular implementation

For the regular and regular+ implementations, it is possible to evaluate the number of elementary operations with the number of tested variables, which each corresponds to a call to the *findBestPosition*() procedure (see Section 4.3) and, therefore, $n-1$ elementary operations. We know that the BI-reg operator necessarily tests n variables and experimental results suggest that BI-reg+ tests approximately nd variables. Figure 4.3 presents the average number of tested variables per iteration for various instances during a HC using the FVI policy for the regular and regular+ implementations. We observe that only a very small number of variables are tested. For example, the regular implementation tests only between 5 and 6 variables for an instance of size 500, which is barely over 1%. As the size of the problem instances grows larger, the number of tested variables per iteration increases, but the proportion compared to n decreases rapidly. This translates to a significantly lower number of operations performed by FVI in comparison to BI with the regular implementation. Notice that, because of the logarithmic x axis, a straight line in the graph represents a logarithmic function of n. We

observe that the number of tested variables per iteration using the FVI policy is at least in the order of $\log n$, which means that the number of operations per iteration is proportional to $n \log n$. As expected, both implementations test the same number of variables for instances of density 100% and regular+ tests fewer variables for sparse instances.



(a) FVI-reg

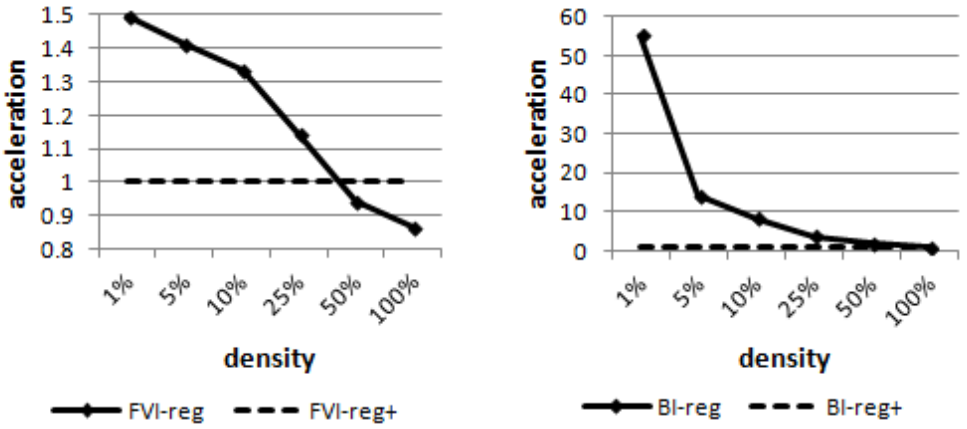


(b) FVI-reg+

Figure 4.3 Average number of tested variables per iteration during a HC using the FVI policy for the regular (Figure a) and regular+ (Figure b) implementations depending on size

For both policies, we have seen that regular+ tests fewer variables than regular. However, there is an additional cost to manage the candidate variables in regular+. Figure 4.4 compares the average execution time of a HC with both implementations by displaying the acceleration gained by using the regular+ operator instead of regular. We observe that regular+ is faster for sparse instances in both implementations and this advantage diminishes as the density grows. With the FVI policy, the regular implementation takes 50% more time for instances of 1% density and is 15% faster for 100% instances. There is a bigger difference with the

BI policy where regular is as much as 55 times slower at $d = 1\%$ and less than 3% faster at $d = 100\%$.



(a) FVI (b) BI

Figure 4.4 Acceleration factor for using regular+ instead of regular with the FVI policy (Figure a) and the BI policy (Figure b), for problem instances of size $n = 1000$

4.6.4 Elementary operations in the tree implementation

A HC iteration using the tree implementation includes deleting an inserting an element in the trees, may involve calls to *locate*(p), and a tree is rebalanced when necessary. Each of these operations takes $\mathcal{O}(nd \log nd)$ operations. As these operations occur when updating the tree after a move, they are independent of the policy. We can therefore estimate the number of elementary operations per HC iteration using the tree implementation at $k \times nd \log_2 nd$, where k is approximately between 2 and 4.

Experimental results find that the average number of elementary operations per iteration is $k' \times nd \log_2 nd$, where $2.2 \leq k' \leq 3.5$. The value of k' is mostly affected by the density of the instance; the lower density yielding the highest k' value. The reduced number of operations at higher densities is due to an implementation optimization described in 4.4.2 that allows to perform the *locate*(p) procedure in constant time when p corresponds to a leaf contained in the tree. This optimization is more effective on dense instances as the trees contain more elements.

While the regular and regular+ implementations benefit from an average number of elementary operations significantly lower than the expected worst case, the tree implementation performs close to its theoretical number of elementary operations.

4.6.5 Slowing factor in the regular implementation

In this section and the next, we measure the slowing factor on elementary operations for different sizes and densities. The main reason a slowing factor may occur is that, as an algorithm routinely uses a larger set of data, the efficiency of the cache memory may decrease, adding latency to many operations. A cache miss occurs when the cpu accesses data that is not currently in the cache memory and needs to wait while it is retrieved from a slower memory component. In our experiments, we measure the number of L1 cache misses.

In Section 4.3, we have presented a modification applied to the regular implementation that uses the antisymmetry property of Equation (4.3). We define regular0 as the implementation before that modification. Note that the regular0 implementation is equivalent to the one used in FVI-reg-04.

Figure 4.5 displays the slowing factor for the regular and regular0 implementation for increasing values of n. We observe that the slowing factor only varies by 20% for the regular implementation. On the other hand, the slowing factor for the regular0 implementations increases rapidly for instances larger than 500. The number of cache misses measured during the execution is consistent with these results, being almost constant with regular and significantly higher with regular0 for large instances. An effect of this slowing factor is visible in Figures 4.1a and 4.1b where the difference between FVI-reg and FVI-reg-04 increases significantly from $n = 500$ to $n = 2000$. Note that the slowing factor is equivalent for the regular and regular+ implementations with both policies because it only applies inside the *findBestPosition*(p) procedure.

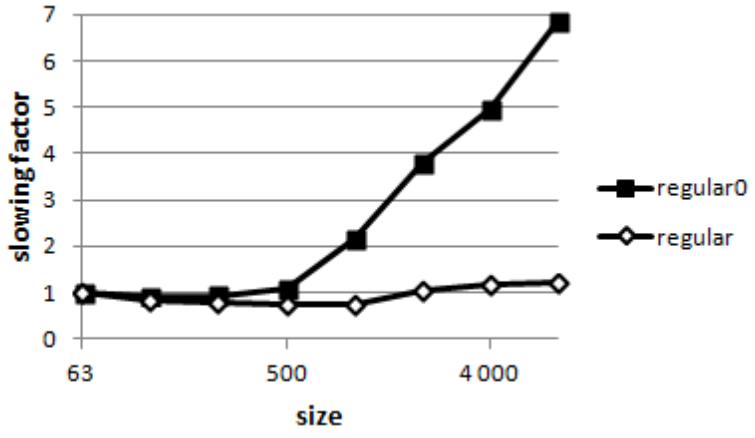


Figure 4.5 Slowing factor for the regular and regular0 implementations depending on size

4.6.6 Slowing factor in the tree implementation

Figure 4.6 presents the slowing factor experienced for each problem instance for the tree implementation. We can see that the slowing factor increases slowly when n is small, then rises rapidly after the instances reach a certain size; this threshold is smaller for dense instances. For larger instances, the slowing factor for all densities increases at approximatively the same rate. We also observe higher values than for the regular implementation. Elementary operations are 14 times slower for the largest and densest instances. To confirm these results, the slowing factor was also calculated from our original experiments from the equation $slowF = cpuHC/nbOp$. The resulting values were very similar. Finally, we measure significantly more cache misses with the tree implementation as the size or density of the problem instance increases, supporting our assumption that the cache memory has an important impact on the execution time.

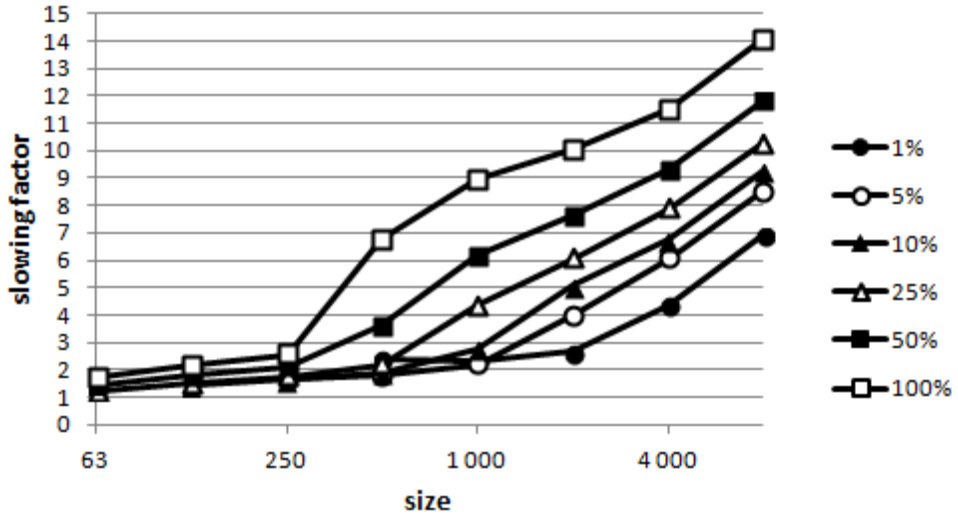


Figure 4.6 Slowing factor for the tree implementation, for all problem instances

4.6.7 Comparing the implementation techniques

In this section, we compare FVI-reg+ and BI-tree, the best performing HC operators for the regular+ and tree implementations respectively. Figure 4.7 shows how much longer BI-tree takes to reach a local optimum compared to FVI-reg+ for all problem instances. It is clear that FVI-reg+ dominates BI-tree for all instances because the time ratio is greater than 1. Also, the density has a much bigger impact than the size on this comparison. The denser the instance, the bigger the advantage for FVI-reg+. However, for a fixed density and instances

larger than 1000, there is very little change in the relative efficiency of the two variants.

In Section 4.6.6, we observed a large slowing factor for the tree implementation. Even if we imagine a perfect scenario where there is no slowing factor, BI-tree only gets faster than FVI-reg+ for problem instances of 1% density and very large instances of 5% density.

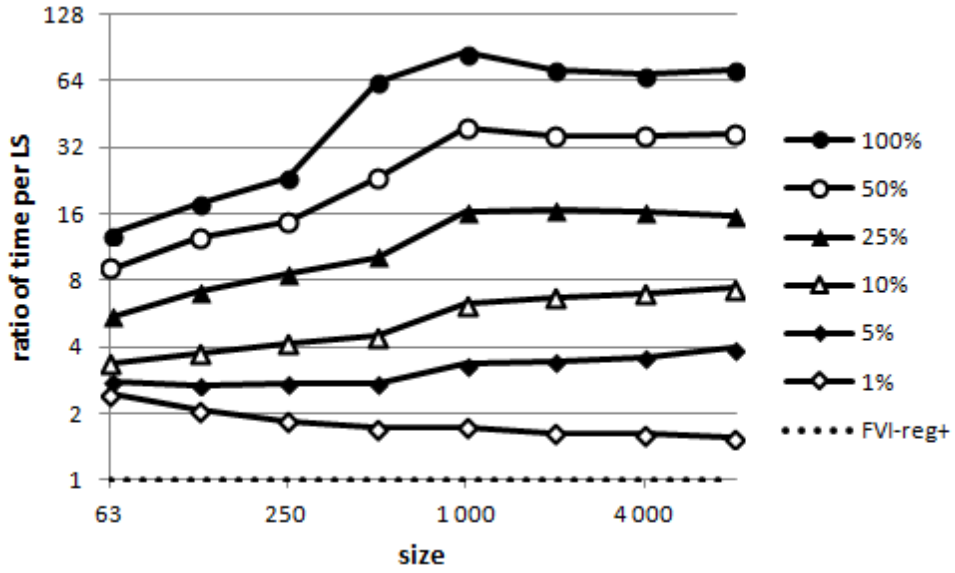


Figure 4.7 Ratio of the average execution time of a HC with BI-tree compared to FVI-reg+ in the context of HCR, for all problem instances

4.6.8 Hill climbing in a memetic algorithm

While the previous sections only consider HC in the context of HCR, we now make observations in the context of MA.

Figure 4.8 displays a ratio of the average time to reach a local optimum with the BI-tree operator compared to the time with FVI-reg+ in the context of a memetic algorithm for all problem instances. Once again, FVI-reg+ is always faster than BI-tree and gets a bigger advantage for denser instances. However, the difference is smaller than in HCR, especially for large and sparse instances. For the instances $(n, d) = (4000, 1\%)$, they differ by as little as 3.5% and they take the same time for the instances $(n, d) = (8000, 1\%)$. In the following, we perform a more detailed analysis of FVI-reg+ and BI-tree in MA to understand where this variation comes from.

The difference between a HC in the two contexts is that the starting configurations in MA have a better score than the ones in HCR. A HC starting from a partially improved solution

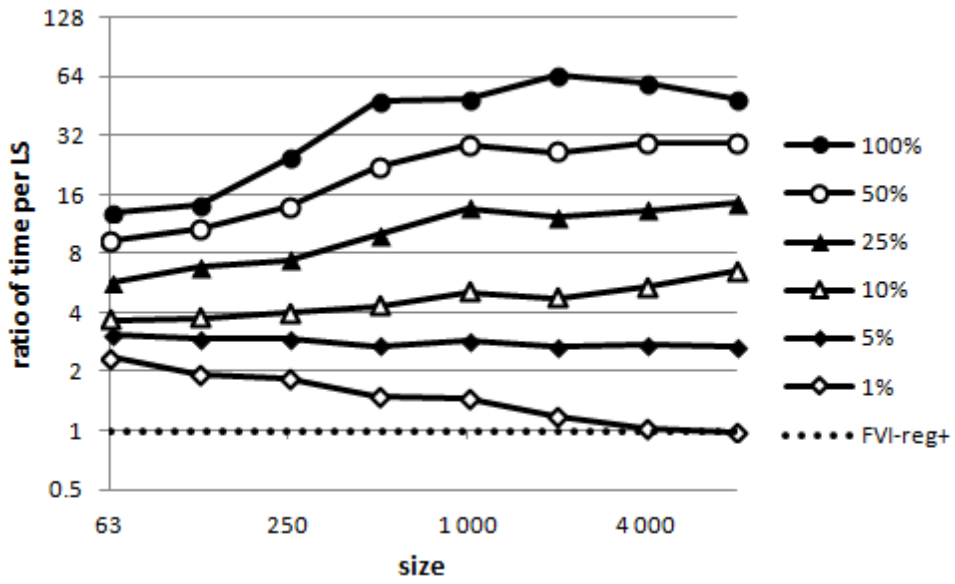


Figure 4.8 Ratio of the average execution time of a HC with BI-tree compared to FVI-reg+ in the context of MA, for all problem instances

will perform less iterations than when it starts from a randomly generated one. This affects both variants (FVI-reg+ and BI-tree) differently.

The tree implementation needs an initialization that is independent from the solution quality. In HCR, the number of iterations is generally large enough that the initialization time is not significant. In MA, the reduced number of iterations gives the initialization more importance.

To understand what changes for the FVI-reg+ operator in a MA, we measured and reported in Figure 4.9 the average number of tested variables per iteration in a HC using the FVI policy at every generation of the memetic algorithm for an instance $(n, d) = (250, 100\%)$. The graph displays that the number of tested variables per iteration increases as the algorithm progresses. In the first few generations, which include the initialization of the population from random configurations, there is a rapid increase, followed by a steady increment of approximately one more tested variable per iteration every 25 generations. During the initialization, the searches on random solutions are similar to the ones in HCR, testing on average 6.4 variables per iteration. After the 213 generations of MA, the HC tests on average 15.7 variables per iteration, which is more than 2.4 times higher. During the full MA run, the average number of tested variables per iteration equals 10. This indicates that each iteration is on average 1.6 times more expensive in MA than in HCR for this problem instance.

Figure 4.10 displays the number of variables tested at every iteration of a HC using FVI on the same problem instance $(n, d) = (250, 100\%)$ starting with a random solution. From

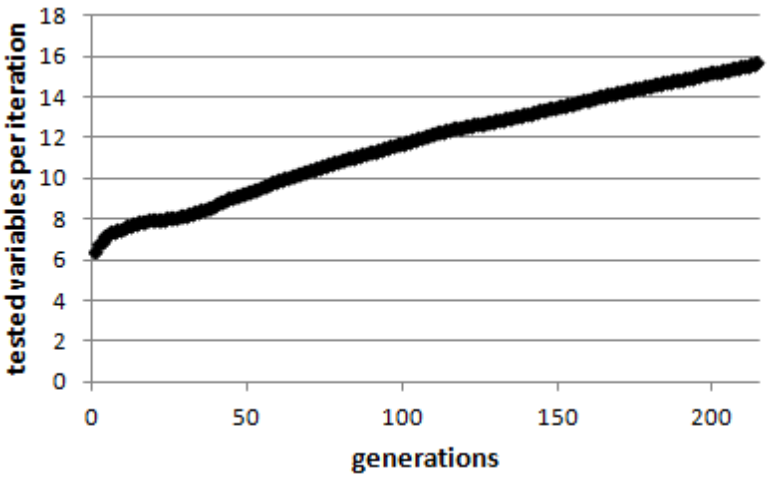


Figure 4.9 Number of tested variables per iteration of FVI during MA depending on the number of generations, for an instance $(n, d) = (250, 100\%)$

the figure, we can observe that, at the beginning of the search, a very small number of variables are tested. Indeed, iteration 325 (more than 50% of the 613 for the whole HC) is the first one to test more than 5% of the variables. The number of tested variables increases significantly only at the very end. On average, 50% of the variable tests are performed in the last 10% of iterations. It is now easier to understand that for the FVI policy, while starting a HC with a better solution reduces the number of iterations required, only the first (and fastest) iterations are skipped. Therefore, the average time and number of tested variables per iteration are both higher.

4.6.9 Comparison with the tree implementation in the literature

In [41], Sakuraba and Yagiura have proposed the original tree implementation and a HC operator that uses this implementation while applying the BI policy. We refer this operator as BI-tree-10. In the paper, the authors present experiments that compare BI-tree-10 with two other HC operators inspired by [43] that implement a best improvement (BI) policy and a first improvement (FI) policy. We note these two operators BI-reg-10 and FI-reg-10 respectively. The conclusion of these experiments is that BI-tree-10 is significantly faster than the other operators.

Figure 4.11 reports the time to reach a local optimum for problem instances of size $n = 2000$ at various densities using six different operators. The figure displays, for the BI-tree-10, BI-reg-10 and FI-reg-10 operators, the running times reported in [41] and, for our operators (BI-tree, BI-reg, FVI-reg), the running times measured with the same problem instances on

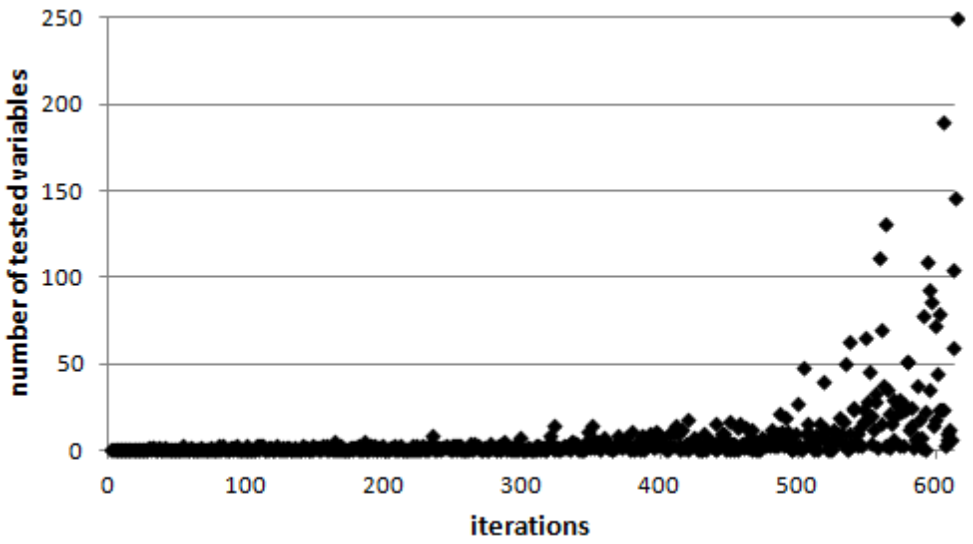


Figure 4.10 Number of tested variables per iteration during HCR using FVI, for an instance $(n, d) = (250, 100\%)$

our computer. Note that our computer is more recent and that it runs a cpu at a 13% faster frequency.

From the figure, we observe that the cpu time reported for our BI-tree is approximately 4 times lower than the one of BI-reg-10. Considering the differences between the two computers, it is plausible that both operators are equally efficient. We also observe that the cpu time reported for BI-reg-10 is around 50 times higher than for our BI-reg. Such a difference indicates that BI-reg-10 is less efficient than BI-reg, which can be at least partially explained by the absence of slowing factor in our operator. There is no clear description of the FI-reg-10 operator in [41]; as it is labeled as first improvement, we can assume it does not apply the FVI policy. Also, the cpu time reported for this operator is three orders of magnitude higher than for FVI-reg.

4.6.10 Summary and analysis of the results

The results of our experiments reveal many interesting findings. Some of these are related to the regular implementation. First, when the FVI policy is used, only a small number of variables are tested, making the operator much faster than with the BI policy. Moreover, and most interestingly, this number grows very slowly with the increase of the size n of the instance. As a result, the average running time of an iteration appears to have a complexity only a little bit larger than $n \log n$. Also, our regular+ implementation makes it possible to

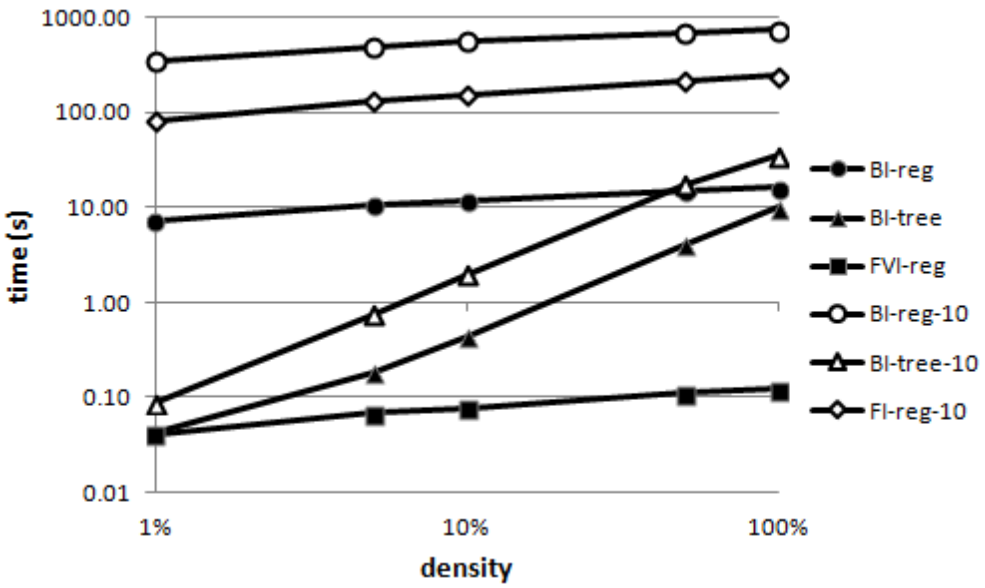


Figure 4.11 Time to reach a local optimum, including those reported by [41], for problem instances of size $n = 2000$

speed up significantly the FVI-reg operator when dealing with sparse instances by reducing further the number of tested variables. Finally, the predominant elementary operation is not executed in constant time; instead, it is affected by a slowing factor due to the use of the cache memory. When using the original implementation [43], this slowing factor is important for instances of size 500 and more. The simple technique we propose practically removes this slowing factor. Note however that the original implementation was designed for targeting traditional problem instances that are both not too large ($n \leq 250$) and generally dense, and it is just fine for this job. The two improvements we propose have a significant effect only for larger and/or sparser instances.

Some other of our findings are related to the tree implementation. First, unlike what happens with the regular and regular+ implementations, there is no advantage in using the FVI policy with the tree implementation because their running times per iteration are similar while the FVI policy requires more iterations. Besides, the tree implementation is the fastest implementation when the BI policy is applied (i.e., BI-tree is faster than BI-reg and BI-reg+).

The most salient findings are related to the comparison between the two main contenders: the best regular+ operator (FVI-reg+) and the best tree operator (BI-tree). First, whatever the size of the instance, the BI-tree operator is much slower than the FVI-reg+ operator when dealing with dense instances, while it is still slower for our sparsest instances, but only by a small margin. Note however that the results tend to be a little bit more flattering for

BI-tree when the HC operator is used within a hybrid algorithm than in the context of HCR.

We can observe that the tree implementation benefits from two sophistications that also reflect on its worst-case complexity – see Section 4.4.3. First, it avoids testing the zero elements in the input matrix which is expected to be helpful for sparse instances. Second, it uses a tree data structure to reduce further the complexity which is expected to better deal with large instances. As a result, it seems counter-intuitive to observe that the best tree operator is not able to catch up the best regular+ operator. Let us distinguish two different cases. First, if we consider problem instances having the same size, we observe that the running time of the BI-tree operator actually decreases much faster than the one of the FVI-reg+ operator when the density decreases. Unfortunately, it starts (for $d = 100\%$) with a huge disadvantage as it is much slower than its opponent – two orders of magnitude slower for $n = 2000$. Second, if we consider problem instances having the same density, we observe that the ratio of the cpu time of the two operators remains stable as the size grows. The reason is that, although the BI-tree operator benefits (moderately) from a smaller increase in the number of iterations and a (much more substantial) advantage with respect to worst-case complexity, this is in a large margin compensated by the decrease in the proportion of variables tested by the FVI-reg+ operator. In addition, the tree operator is hampered by a growing slowing factor.

Finally, we can notice that the BI-tree operator is complicated to implement and that it never dominates the FVI-reg+ operator in our experiment. In spite of that, we believe it is much too early to dismiss it. First, it may outrun FVI-reg+ when dealing with instances that are still sparser than our sparsest instances, as it is the case in the benchmark used for FASP [39]. Also, the tree implementation may benefit from further improvement in computer technology that could eventually reduce its slowing factor, for example, with a better management of the cache memory. In addition, as the tree implementation is the most efficient for the BI policy, it seems promising in the context of adapting the tabu metaheuristic to the LOP because, unlike HC, this metaheuristic requires to explore a large portion of the neighborhood – as it is often suggested to apply the best (non-tabu) move on each iteration of a tabu algorithm [20].

4.7 Conclusion

Our paper is concerned with the development of efficient local search heuristics for the linear ordering problem and it is more specifically focused on hill climbing (HC) operators that exploit the insert neighborhood. There are three different types of contributions in the paper. First, we have presented and analyzed neighborhood implementation techniques proposed in

the literature. The detailed description we make of the different techniques will be helpful for anyone who wants to develop a hybrid heuristic to the LOP. In particular, we have presented a new version of the tree implementation that we believe is easier to understand than the original version.

Second, we have proposed several improvements related to the regular implementation technique that make it remarkably competitive when dealing with large and sparse LOP instances. In practice, a memetic algorithm such as the one proposed by Schiavinotto et Stützle in [43] equipped with a HC operator dotted with the implementation we propose (namely the regular+ implementation) would be both reasonably easy to implement and very efficient for dealing with sparse LOP instances (such as FASP instances) that are present in several real-life applications.

Finally, we have performed an extensive experimental study related to the influence of the HC operator's features (policy and implementation technique) on the running time of the algorithm, depending on the size and the density of the instance. This study provides information that may help develop new more efficient LOP neighborhood search heuristics in the future.

CHAPITRE 5 RÉSULTATS COMPLÉMENTAIRES

5.1 Détails d'implémentation

Une grande importance est accordée aux techniques d'implémentation des opérateurs de recherche dans l'article présenté au chapitre 4. Ainsi, le développement des nouvelles techniques proposées est détaillé dans cette section.

Structures de données

Dans ce travail, on présente une implémentation arborescente qui a été développée de façon indépendante, bien qu'une version similaire, utilisant aussi des arbres, ait été proposée auparavant, mais est passée inaperçue lors de la revue de littérature initiale. La section 4.4 décrit cette implémentation, la structure de données nommée MPS et la distinction entre les deux variantes d'implémentation arborescente. La structure de données MPS est basée sur une relation de récurrence observée lors de l'exploration du voisinage d'insertion du LOP et est implémentée à l'aide d'un arbre binaire équilibré de type AVL. Cette structure arborescente est choisie pour sa simplicité et, surtout, car la hauteur de l'arbre est généralement plus faible que pour d'autres types d'arbre équilibré tel l'arbre rouge et noir, par exemple. On remarque que, durant la recherche locale, le nombre de mises à jour de nœuds et de recherches d'une feuille est plus élevé que le nombre d'opérations d'insertion et de suppression. Ainsi, il est favorable de conserver un arbre plus petit pour avantager les opérations qui dépendent de la hauteur de l'arbre, même si les modifications à la structure sont plus lentes pour mieux maintenir l'équilibre.

Gestion de la mémoire

La gestion de la mémoire peut avoir un impact considérable sur l'exécution d'un algorithme. En particulier, on observe une influence énorme sur le facteur de ralentissement, qui est analysé aux sections 4.6.5 et 4.6.6. À la section 4.3, on présente une simple modification de l'implémentation quadratique qui permet d'accélérer significativement l'exécution comparativement à la version décrite dans la littérature en effectuant les accès mémoire de manière plus efficace grâce à la propriété à l'équation (4.3). Bien sûr, cette propriété est aussi utilisée pour tous les autres opérateurs. Par contre, dans l'implémentation arborescente, elle n'a qu'un impact marginal puisqu'elle peut seulement être appliquée à l'initialisation. Il y a tout de même une optimisation à mentionner pour la structure de données MPS qui est utilisée

dans cette implémentation. En effet, on remarque que la taille de chaque arbre créé reste constante lors de la descente. Il est donc possible d'allouer une seule fois, à l'initialisation, un espace mémoire contigu pour contenir l'arbre en entier et ainsi accélérer les accès mémoire. Cette simple modification rend l'exécution jusqu'à deux fois plus rapide.

5.2 Tests réalisés avec d'autres jeux de données

Lors des expériences présentées à la section 4.6, seuls des exemplaires générés aléatoirement sont utilisés, car ils ont tous une taille et une densité spécifiquement choisies pour faciliter l'analyse des résultats. Dans cette section, les opérateurs BI-tree et FVI-reg+ sont évalués sur des exemplaires basés sur des données réelles (jeu de données XLOLIB) et les résultats sont comparés à ceux obtenus avec le jeu de données aléatoire que nous avons généré.

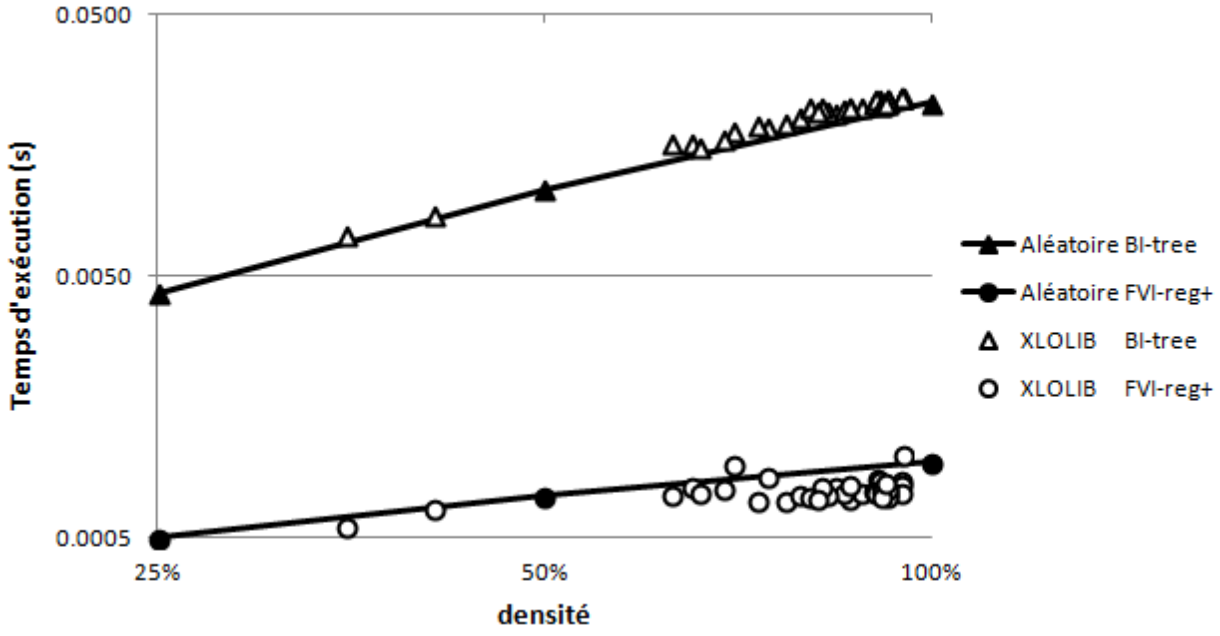


Figure 5.1 Temps d'exécution d'une descente, pour les exemplaires du problème de taille $n = 250$ générés aléatoirement et ceux tirés de XLOLIB

La figure 5.1 affiche le temps d'exécution d'une descente utilisant les opérateurs BI-tree et FVI-reg+ à partir d'une configuration initiale aléatoire. Les exemplaires de taille $n = 250$ sont utilisés, car ce sont les seuls communs aux deux jeux de données. Une droite représente les résultats obtenus avec les exemplaires aléatoires et leur interpolation linéaire pour toutes densités. Les temps d'exécution pour les exemplaires de XLOLIB sont chacun représentés par

un point à la densité correspondante. On observe que le temps pour atteindre un optimum local est semblable pour les deux types d'exemplaires. On remarque tout de même que, pour une densité fixe, l'opérateur BI-tree a tendance à être légèrement plus efficace pour les données aléatoires que pour les données réelles, alors que c'est généralement le contraire pour FVI-reg+. Cette variation est relativement minime comparée à la différence entre les deux opérateurs. On peut donc s'attendre à ce que les résultats présentés pour la comparaison et l'analyse des opérateurs de recherche se généralisent à d'autres types de jeu de données, dont les jeux de données réelles.

5.3 Utilisation du voisinage réduit

Un voisinage réduit est proposé par Ceberio et al. [7] dans lequel une analyse préliminaire permet d'ignorer certains mouvements d'insertion lors de la recherche locale. Ces mouvements, qui consistent à insérer un élément à une position où il ne peut en aucun cas générer d'optimum local, sont appelés des mouvements interdits. Évidemment, plus la proportion de mouvements interdits est élevée, plus cette méthode est efficace. Les résultats fournis dans [7] indiquent que, à cause de l'analyse initiale qui prend beaucoup de temps, cette méthode est seulement utile pour des exemplaires dont la proportion de mouvements interdits est supérieure à 15%. À partir de ces résultats, on remarque aussi que la proportion de mouvements interdits semble diminuer lorsque la taille des exemplaires augmente. Par contre, il n'y a pas d'information sur l'effet de la densité sur cette méthode. Nous avons donc implémenté l'algorithme décrit dans [7] pour mesurer la proportion de mouvements interdits sur les exemplaires de taille et densité variées que nous avons générés. Cette implémentation est validée en comparant la proportion de mouvements interdits que nous obtenons pour les jeux de données LOLIB et XLOLIB avec les résultats affichés dans un graphique de l'article [7].

La figure 5.2 affiche la proportion de mouvements interdits par le voisinage réduit pour des exemplaires de taille et densité variées. On observe que, comme prévu, cette proportion diminue quand la taille augmente. De plus, cette proportion est plus faible pour les exemplaires peu denses. On peut donc en conclure que l'utilisation du voisinage réduit est plus appropriée seulement pour les petits exemplaires denses. En comparant ces résultats sur d'autres jeux de données, on remarque aussi que, pour une taille et une densité fixes, la proportion de mouvements interdits est plus faible pour les exemplaires aléatoires que pour ceux provenant de données réelles dans LOLIB et XLOLIB. Dans ce mémoire, on s'intéresse plus à la performance des opérateurs de recherche appliqués sur des exemplaires de grande taille et de toute densité. Il est donc préférable de ne pas utiliser le voisinage réduit dans ce cas.

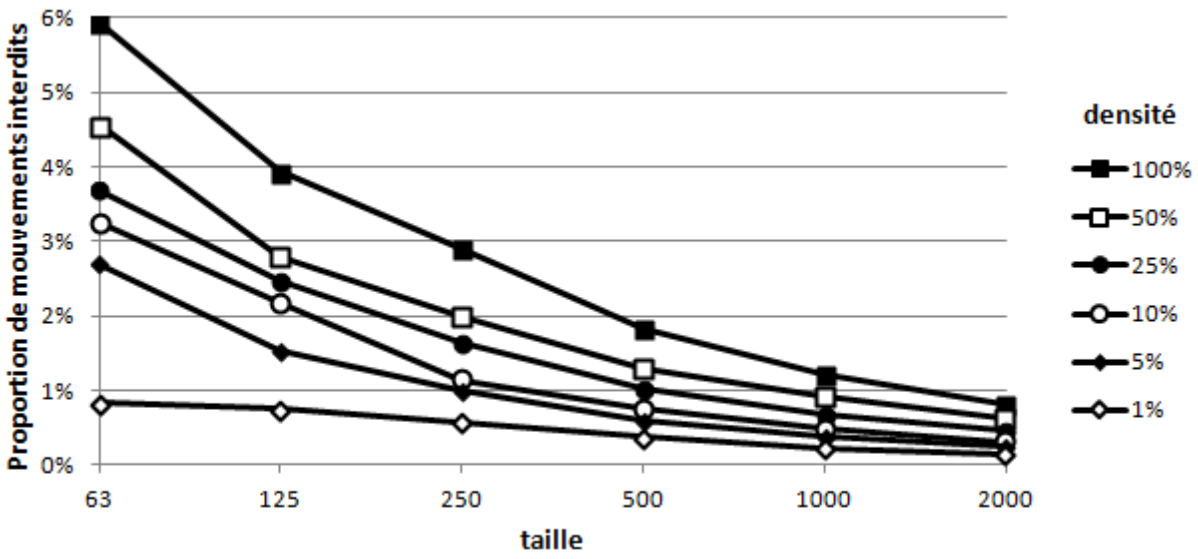


Figure 5.2 Proportion de mouvements interdits par le voisinage réduit, pour plusieurs exemplaires de taille et densité variées

5.4 Précisions sur l'analyse détaillée

Cette section fait un retour sur les résultats de l'analyse présentée dans l'article, à la section 4.6.10. En particulier, le modèle décrit à la section 4.5.3 est légèrement modifié afin de mieux présenter le comportement asymptotique des opérateurs de recherche. Dans le modèle de base, le temps d'exécution d'une descente *cpuHC* est représenté par le produit du nombre d'itérations *nbIt*, du temps d'exécution de référence d'une opération élémentaire *cpuOpRef*, du nombre d'opérations élémentaires *nbOp* et du facteur de ralentissement *slowF*. On décompose maintenant *nbOp* en un nombre d'opérations élémentaires attendues en pire cas *nbOpWC* et la proportion d'opérations réellement effectuées *propOp* tel que $nbOp = nbOpWC \times propOp$. La valeur de *nbOpWC* dépend de l'implémentation utilisée. Pour l'implémentation quadratique (reg), $nbOpWC = n \times (n-1)$ et pour l'implémentation arborescente (tree), $nbOpWC = 4 \times nd \log nd$. Ainsi, on obtient la formule suivante pour représenter le temps d'exécution d'une descente.

$$cpuHC = nbIt \times cpuOpRef \times nbOpWC \times propOp \times slowF$$

On s'intéresse alors à l'impact d'augmenter la taille ou la densité d'un exemplaire sur chacune des composantes du modèle. D'abord, on évalue l'opérateur FVI-reg+ dans une descente sur

deux exemplaires et on note, pour chaque composante, x_1 et x'_1 les valeurs obtenues. On répète cette étape pour l'opérateur BI-tree en notant les valeurs x_2 et x'_2. On calcul ensuite l'impact observé en changeant d'exemplaire pour chaque opérateur par le ratio $R_i = \frac{x'_i}{x_i}, i \in \{1, 2\}$.

Les tableaux 5.1 et 5.2 présentent la décomposition du temps d'exécution d'une descente à l'aide de ce modèle en variant respectivement la taille et la densité de l'exemplaire traité. De plus, le ratio R_2/R_1 est affiché afin d'indiquer, pour chaque composante du modèle, quel opérateur voit un avantage à augmenter la taille ou la densité de l'exemplaire. Une valeur supérieure à 1 signifie que FVI-reg+ est avantagé alors que BI-tree l'est pour une valeur inférieure à 1. Par exemple, à la ligne *nbIt* du tableau 5.1, on observe aux colonnes R_1 et R_2 que, lorsque la taille de l'exemplaire passe de 500 à 8000, le nombre d'itérations est multiplié par 30.7 pour FVI-reg+ et seulement par 25.7 pour BI-tree. Enfin, deux valeurs sont présentées pour le temps d'exécution de la descente. La première, $cpuHC_m$, est le temps obtenu avec le modèle en multipliant chacune des rangées au-dessus. Dans la colonne de droite, le ratio des deux opérateurs est représenté par son logarithme en base 2, permettant ainsi de mieux comparer les différentes valeurs puisque $cpuHC_m$ est alors la somme des rangées au-dessus. La deuxième valeur du temps d'exécution, $cpuHC_r$, est le temps réel, mesuré durant l'expérimentation, et est affichée comme référence. Il faut noter que le modèle n'est pas construit pour calculer précisément le temps d'exécution, mais plutôt pour isoler certaines composantes. On peut donc s'attendre à ce que les deux valeurs diffèrent.

Le tableau 5.1 affiche l'impact de multiplier par 16 la taille de l'exemplaire traité. On observe que BI-tree est principalement avantagé par la complexité en pire cas *nbOpWC*, mais cela est presque entièrement compensé par la proportion d'opérations élémentaires réellement effectuées *propOp* qui diminue énormément pour FVI-reg+. De plus, les impacts sur le nombre d'itérations *nbIt* et sur le facteur de ralentissement *slowF* sont opposés et beaucoup plus faibles que pour *nbOpWC* et *propOp*. Au final, les différentes composantes du modèle s'annulent presque entre elles et on observe seulement un faible avantage théorique pour BI-tree lorsque la taille des exemplaires augmente.

Le tableau 5.2 affiche l'impact de diviser par 4 la densité de l'exemplaire traité en changeant de 100% à 25%. Dans ce cas, l'opérateur FVI-reg+ n'est pas affecté par *nbOpWC* ni *slowF* alors que BI-tree est avantagé par les deux puisqu'il s'adapte à la densité. L'effet de *propOp* et *nbIt* désavantage légèrement BI-tree, mais pas assez pour compenser les autres facteurs. Il en résulte que BI-tree est énormément accéléré par la diminution de la densité alors que FVI-reg+ l'est seulement un peu. Cela aide à expliquer pourquoi BI-tree est compétitif sur les exemplaires très peu denses, même s'il est beaucoup plus lent que FVI-reg+ sur les exemplaires denses.

Tableau 5.1 Décomposition du temps d'exécution d'une descente sur des exemplaires de taille $n = 500$ et $n = 8000$ avec une densité $d = 100\%$

	$n = 500$		$n = 8000$					
	x_1	x_2	x'_1	x'_2	R_1	R_2	$\frac{R_2}{R_1}$	$\log_2 \frac{R_2}{R_1}$
nbIt	1261.1	792.5	38676.1	20370.1	30.7	25.7	0.84	-0.25
cpuOpRef	9.16E-10	4.71E-09	9.16E-10	4.71E-09	1	1	1.00	0.00
nbOpWC	249500	17932	63992000	414905	256	23	0.09	-3.47
propOp	0.0118	0.5572	0.0014	0.5437	0.12	0.98	8.33	3.06
slowF	0.76	6.75	1.21	14.10	1.58	2.09	1.32	0.40
$cpuHC_m$	0.00260	0.2519	3.776	305.4	1455	1212	0.83	-0.26
$cpuHC_r$	0.00401	0.2544	4.510	322.9	1123	1269	1.13	0.18

Tableau 5.2 Décomposition du temps d'exécution d'une descente sur des exemplaires de taille $n = 500$ avec une densité $d = 100\%$ et $d = 25\%$

	$d = 100\%$		$d = 25\%$					
	x_1	x_2	x'_1	x'_2	R_1	R_2	$\frac{R_2}{R_1}$	$\log_2 \frac{R_2}{R_1}$
nbIt	1261.1	792.5	993.9	679.5	0.79	0.86	1.09	0.12
cpuOpRef	9.16E-10	4.71E-09	9.16E-10	4.71E-09	1	1	1.00	0.00
nbOpWC	249500	17932	249500	3483	1.00	0.19	0.19	-2.36
propOp	0.0118	0.5572	0.0102	0.7433	0.86	1.33	1.54	0.63
slowF	0.76	6.75	0.76	2.19	1.00	0.32	0.32	-1.63
$cpuHC_m$	0.00260	0.2519	0.00177	0.01813	0.68	0.07	0.11	-3.24
$cpuHC_r$	0.00401	0.2544	0.00264	0.02690	0.66	0.11	0.16	-2.64

CHAPITRE 6 DISCUSSION GÉNÉRALE

Ce chapitre présente un retour sur les résultats obtenus dans les chapitres 4 et 5 en soulignant leur correspondance avec les objectifs définis à la section 1.3. Nous revenons ensuite sur l'analyse des opérateurs de recherche.

6.1 Retour sur les résultats

Dans ce travail, plusieurs techniques d'implémentation sont décrites et appliquées dans des opérateurs de recherche avec les politiques BI et FVI. Premièrement, nous avons développé une implémentation nommée *tree* qui est une version plus claire de la technique d'implémentation arborescente déjà présentée dans la littérature. Ces deux versions s'exécutent avec la plus faible complexité connue pour explorer le voisinage d'insertion. Ensuite, l'implémentation quadratique, qui est régulièrement utilisée dans la littérature, est optimisée pour l'accélérer sur de gros exemplaires et notée *regular*. Enfin, nous avons implémenté *regular+*, qui est basé sur *regular* et est avantagée sur les exemplaires peu denses. Pour chacune de ces techniques d'implémentation, l'opérateur de recherche le plus efficace est respectivement BI-tree, FVI-reg et FVI-reg+.

Une analyse détaillée de tous les opérateurs décrits est ensuite effectuée en incluant aussi ceux retrouvés dans la littérature. Celle-ci révèle que malgré une plus faible complexité dans le pire cas, le meilleur opérateur utilisant l'implémentation arborescente (BI-tree) est tout de même plus lent que le plus efficace pour chacune des deux autres implémentations proposées (FVI-reg et FVI-reg+). En effet, l'opérateur FVI-reg est le plus efficace sur les exemplaires denses alors que FVI-reg+ domine pour les densités faibles. Par contre, la technique d'implémentation avec des arbres est supérieure pour appliquer la politique de recherche de la meilleure amélioration (BI). La section 4.6.10 contient plus de détails sur la comparaison et l'analyse des divers opérateurs de recherche locale.

Enfin, les trois opérateurs de recherche mentionnés améliorent donc ceux de la littérature dans différentes situations, ce qui répond au premier objectif de ce travail. Aussi, l'analyse effectuée sur les différents opérateurs permet de préciser l'effet de plusieurs facteurs dans la descente. Par exemple, l'étude du facteur de ralentissement a confirmé que la gestion de la mémoire peut avoir un impact significatif sur des exemplaires plus gros et ainsi, a pu orienter l'optimisation des implémentations tel que décrit à la section 5.1. Finalement, cette analyse remplit directement le second objectif en comparant les nouveaux opérateurs à ceux dans la

littérature et fournit l'information nécessaire pour guider le choix d'un opérateur de recherche dans différentes situations tel qu'attendu pour le dernier objectif mentionné dans ce travail. Plus de détails pour la sélection d'un opérateur sont présentés à la section 7.2.

6.2 Retour sur l'analyse des opérateurs de recherche

Une composante majeure du travail consiste en une analyse détaillée des différents opérateurs de recherche utilisant le voisinage d'insertion, tant ceux existants dans la littérature que les nouveaux proposés. Pour ce faire, un modèle est défini pour isoler l'effet de différents facteurs affectant le temps d'exécution d'une descente et mettre en évidence l'impact d'implémentation et de la politique de recherche utilisé par un opérateur. Ce modèle est très simplifié, en particulier car un seul type d'opérations élémentaires est considéré pour chaque technique d'implémentation. Par contre, il est adéquat dans notre étude puisque son but est de comprendre le comportement des opérateurs et non pas de prédire leur temps d'exécution.

L'analyse est surtout concentrée sur le temps d'exécution des opérateurs de recherche dans le contexte de descentes avec relances à partir de configurations aléatoires. Ce contexte simplifie l'analyse puisque chaque descente est indépendante des autres et aucun paramètre n'est nécessaire pour ajuster l'algorithme. De plus, bien que les solutions trouvées par une descente ne changent pas selon la technique d'implémentation utilisée, la politique de recherche peut les influencer. On observe par contre que le score des solutions obtenues avec les différents opérateurs de recherche ne varie pas significativement. Dans le contexte de descentes avec relance, le temps d'exécution est alors un bon indicateur de la performance de l'opérateur tandis que dans le cadre d'une heuristique hybride, différents autres facteurs peuvent influencer les résultats de chaque algorithme. Une analyse des opérateurs dans un algorithme mémétique aborde donc quelques-unes des distinctions entre les deux contextes qui peuvent affecter le temps d'exécution de la descente.

Bien que seuls des exemplaires aléatoires soient utilisés dans l'analyse présentée, d'autres expériences ont été réalisées à la section 5.2 avec un banc de test basé sur des données réelles (XLOLIB). On y observe des résultats similaires, indiquant donc que l'étude peut être généralisée à d'autres types d'exemplaires du problème, notamment les jeux de données réelles.

CHAPITRE 7 CONCLUSION ET RECOMMANDATIONS

Ce chapitre résume d'abord les principaux résultats du travail et rappelle les contributions apportées. Ensuite, nous proposons des recommandations pour choisir un opérateur de recherche adéquat selon le type d'exemplaires à traiter. Enfin, on présente les limitations des résultats ainsi que de nouvelles voies de recherche.

7.1 Synthèse des travaux

Les contributions de ce mémoire sont principalement axées sur le développement d'heuristiques de recherche locale efficaces pour le LOP. Premièrement, nous avons repris l'implémentation quadratique en présentant une simple modification qui amène une meilleure gestion de la mémoire, permettant de traiter de gros exemplaires. Une autre amélioration est apportée pour adapter cette implémentation aux exemplaires peu denses. De plus, une nouvelle version de la technique d'implémentation arborescente est proposée à partir d'une abstraction de l'exploration du voisinage, ce qui la rend plus claire et facile à manipuler. De plus, nous avons remarqué que l'implémentation d'un opérateur de recherche peut affecter son temps d'exécution par un facteur allant jusqu'à deux ordres de grandeur. Les implémentations que nous avons décrites présentent toutes un avantage par rapport à celle dans la littérature.

Deuxièmement, nous avons décrit et comparé les divers opérateurs de recherche utilisant les implémentations décrites avec les politiques de recherche BI et FVI, ce qui permet de guider le choix d'un opérateur dans une heuristique hybride et donne des conseils pour son implémentation.

Ensuite, une étude expérimentale mesure l'effet de l'implémentation et de la politique de recherche utilisée par les opérateurs sur le temps d'exécution de la descente. Cette étude permet d'expliquer les résultats de la comparaison des divers opérateurs et ainsi peut être utile pour le développement de nouvelles méthodes d'exploration du voisinage.

Enfin, nous présentons à la section 7.2 des recommandations pour choisir le meilleur opérateur de recherche en fonction du type d'exemplaires à traiter. En particulier, l'opérateur FVI-reg+ que nous avons proposé est le plus efficace pour résoudre des exemplaires du problème FASP.

7.2 Recommandations

Premièrement, pour les exemplaires denses, typiquement associés au LOP, l'opérateur FVI-reg est le plus efficace. C'est d'ailleurs celui-ci qui est généralement utilisé dans les algorithmes récents dans la littérature tel l'algorithme mémétique présenté dans [43]. Dans ce cas, il peut aussi être avantageux, surtout pour les petits exemplaires, de considérer le voisinage réduit [7] dépendant de la proportion de mouvements interdits.

Deuxièmement, pour les exemplaires de grande taille et peu denses, typiquement associés au FASP, c'est plutôt l'opérateur FVI-reg+ qui est le plus efficace lors de la descente. Sur des exemplaires de densité très faible, il est possible que l'opérateur BI-tree prenne l'avantage, spécialement s'il est utilisé dans une heuristique hybride.

De plus, dans le cas où la politique de recherche BI doit absolument être appliquée pour l'heuristique choisie, l'implémentation arborescente est conseillée avec l'opérateur BI-tree.

Enfin, peu importe l'opérateur choisi, son implémentation aura une influence significative sur sa rapidité. Il est donc important d'y porter attention. En particulier, la propriété 4.3 devrait être utilisée autant que possible afin d'assurer un accès efficace en mémoire tel qu'appliqué sur l'implémentation quadratique à la section 4.3. Finalement, les deux variantes de l'implémentation arborescente offrent des performances similaires, mais celle que nous proposons à la section 4.4 est plus claire et facile à manipuler.

7.3 Limitations de la solution proposée

Dans le but d'isoler le comportement des opérateurs de recherche, l'étude se concentre sur le contexte spécifique de descente à partir de configurations initiales aléatoires. Par contre, cette situation est peu intéressante en pratique puisque la qualité des solutions trouvées n'est pas suffisante. En général, la recherche locale sera plutôt incorporée dans une heuristique hybride. Dans ce cas, le meilleur opérateur peut varier selon le type d'heuristique utilisée ainsi que ses paramètres. De plus, seul le temps d'exécution est pris en compte alors que chaque heuristique peut privilégier certaines propriétés supplémentaires des opérateurs de recherche. L'étude fournit donc des pistes de solutions pour l'implémentation d'heuristique hybride, mais chaque application est un cas particulier qui requiert une analyse spécifique supplémentaire, ce qui est hors de la portée de notre étude.

Une des conclusions de l'analyse est que l'implémentation d'un opérateur de recherche peut avoir un impact significatif sur son temps d'exécution. Ainsi, la comparaison expérimentale des opérateurs est principalement valide pour des implémentations équivalentes à celles que

nous avons utilisées. En particulier, la technique d'implémentation arborescente est plutôt compliquée et de nombreuses optimisations sont possibles, ce qui la rend difficile à reproduire exactement et laisse la possibilité de l'améliorer encore.

7.4 Améliorations futures

Les améliorations futures consistent principalement à utiliser les opérateurs proposés pour implémenter la recherche locale dans des heuristiques hybrides appliquées au LOP dans la littérature. Cela permettrait de les accélérer et d'obtenir les algorithmes les plus efficaces pour le FASP. De plus, l'abstraction MPS et la structure arborescente récursive proposées peuvent être adaptées à différentes heuristiques de recherche locale. En particulier, la recherche tabou semble intéressante, car un tel algorithme proposé dans la littérature est parmi les plus efficaces pour le LOP, malgré l'utilisation de l'implémentation de base, et qu'il est généralement suggéré d'appliquer la politique BI dans la recherche tabou. Une adaptation de l'implémentation arborescente dans cette heuristique est donc susceptible d'obtenir de bonnes performances sur le temps de calcul et la qualité des solutions.

RÉFÉRENCES

[1] H. Aujac, "La hiérarchie des industries dans un tableau des échanges interindustriels : et ses conséquences sur la mise en œuvre d'un plan national décentralisé", *Revue économique*, pp. 169–238, 1960.

[2] O. Becker, "Das helmstädtersche reihenfolgeproblem—die effizienz verschiedener näherungsverfahren", *Computers uses in the social science*, 1967.

[3] K. Boenchendorf, *Reihenfolgeprobleme : Mean-flow-time sequencing.* Hain, 1982.

[4] L. C. Briand, J. Feng, et Y. Labiche, "Using genetic algorithms and coupling measures to devise optimal integration test orders", dans *Proceedings of the 14th international conference on Software engineering and knowledge engineering.* ACM, 2002, pp. 43–50.

[5] V. Campos, M. Laguna, et R. Martí, "Scatter search for the linear ordering problem", *New ideas in optimization*, pp. 331–339, 1999.

[6] V. Campos, F. Glover, M. Laguna, et R. Martí, "An experimental evaluation of a scatter search for the linear ordering problem", *Journal of Global Optimization*, vol. 21, no. 4, pp. 397–414, 2001.

[7] J. Ceberio, A. Mendiburu, et J. A. Lozano, "The linear ordering problem revisited", *European Journal of Operational Research*, vol. 241, no. 3, pp. 686–696, 2015.

[8] S. Chanas et P. Kobylański, "A new heuristic algorithm solving the linear ordering problem", *Computational optimization and applications*, vol. 6, no. 2, pp. 191–205, 1996.

[9] I. Charon et O. Hudry, "Lamarckian genetic algorithms applied to the aggregation of preferences", *Annals of Operations Research*, vol. 80, pp. 281–297, 1998.

[10] ——, "A branch-and-bound algorithm to solve the linear ordering problem for weighted tournaments", *Discrete Applied Mathematics*, vol. 154, no. 15, pp. 2097–2116, 2006.

[11] H. B. Chenery et T. Watanabe, "International comparisons of the structure of production", *Econometrica : Journal of the Econometric Society*, pp. 487–521, 1958.

[12] T. Christof et G. Reinelt, "Algorithmic aspects of using small instance relaxations in parallel branch-and-cut", *Algorithmica*, vol. 30, no. 4, pp. 597–629, 2001.

[13] R. K. Congram, “Polynomially searchable exponential neighbourhoods for sequencing problems in combinatorial optimisation”, Thèse de doctorat, University of Southampton, 2000.

[14] B. Correal et P. Galinier, “On the complexity of searching the linear ordering problem neighborhoods”, dans *Evolutionary Computation in Combinatorial Optimization*. Springer, 2015, pp. 150–159.

[15] J. S. deCani, “A branch and bound algorithm for maximum likelihood paired comparison ranking”, *Biometrika*, vol. 59, no. 1, pp. 131–135, 1972.

[16] C. G. Garcia, D. Pérez-Brito, V. Campos, et R. Martí, “Variable neighborhood search for the linear ordering problem”, *Computers & operations research*, vol. 33, no. 12, pp. 3549–3565, 2006.

[17] M. R. Garey et D. S. Johnson, “Computers and intractability : a guide to the theory of np-completeness. 1979”, *San Francisco, LA : Freeman*, 1979.

[18] F. Glover, “Future paths for integer programming and links to artificial intelligence”, *Computers & operations research*, vol. 13, no. 5, pp. 533–549, 1986.

[19] ——, “A template for scatter search and path relinking”, dans *Artificial evolution*. Springer, 1998, pp. 1–51.

[20] F. Glover et M. Laguna, *Tabu Search*. Springer, 2013.

[21] F. Glover, T. Klastorin, et D. Kongman, “Optimal weighted ancestry relationships”, *Management Science*, vol. 20, no. 8, pp. 1190–1193, 1974.

[22] F. Glover, M. Laguna, et R. Martí, “Fundamentals of scatter search and path relinking”, *Control and cybernetics*, vol. 29, no. 3, pp. 653–684, 2000.

[23] M. Grötschel, M. Jünger, et G. Reinelt, “A cutting plane algorithm for the linear ordering problem”, *Operations research*, vol. 32, no. 6, pp. 1195–1220, 1984.

[24] H. J. F. Huacuja, G. C. Valdez, R. A. P. Rangel, J. G. Barbosa, L. C. Reyes, J. M. C. Valadez, H. J. P. Soberanes, et D. T. Villanueva, “Scatter search with multiple improvement methods for the linear ordering problem”, *Malaysian Journal of Computer Science*, vol. 25, no. 2, 2012.

[25] G. Huang et A. Lim, "Designing a hybrid genetic algorithm for the linear ordering problem", dans *Genetic and Evolutionary Computation—GECCO 2003*. Springer, 2003, pp. 1053–1064.

[26] M. Jünger et P. Mutzel, *2-layer straightline crossing minimization : Performance of exact and heuristic algorithms*. Max-Planck-Institut für Informatik, 1996.

[27] R. Kaas, "A branch and bound algorithm for the acyclic subgraph problem", *European Journal of Operational Research*, vol. 8, no. 4, pp. 355–362, 1981.

[28] R. M. Karp, *Reducibility among combinatorial problems*. Springer, 1972.

[29] J. Kelley, "The cutting-plane method for solving convex programs", *Journal of the Society for Industrial and Applied Mathematics*, pp. 703–712, 1960.

[30] J. G. Kemeny, "Mathematics without numbers", *Daedalus*, vol. 88, no. 4, pp. 577–591, 1959.

[31] B. W. Kernighan et S. Lin, "An efficient heuristic procedure for partitioning graphs", *Bell system technical journal*, vol. 49, no. 2, pp. 291–307, 1970.

[32] D. E. Knuth, D. E. Knuth, et D. E. Knuth, *The Stanford GraphBase : a platform for combinatorial computing*. Addison-Wesley Reading, 1993, vol. 37.

[33] M. Laguna, R. Marti, et V. Campos, "Intensification and diversification with elite tabu search solutions for the linear ordering problem", *Computers & Operations Research*, vol. 26, no. 12, pp. 1217–1230, 1999.

[34] A. H. Land et A. G. Doig, "An automatic method of solving discrete programming problems", *Econometrica : Journal of the Econometric Society*, pp. 497–520, 1960.

[35] R. Martí et G. Reinelt, *The linear ordering problem : exact and heuristic methods in combinatorial optimization*. Springer Science & Business Media, 2011, vol. 175.

[36] R. Martí, G. Reinelt, et A. Duarte, "A benchmark library and a comparison of heuristic methods for the linear ordering problem", *Computational optimization and applications*, vol. 51, no. 3, pp. 1297–1317, 2012.

[37] J. E. Mitchell et B. Borchers, "Solving linear ordering problems with a combined interior point/simplex cutting plane algorithm", dans *High performance optimization*. Springer, 2000, pp. 349–366.

[38] N. Mladenović et P. Hansen, "Variable neighborhood search", *Computers & Operations Research*, vol. 24, no. 11, pp. 1097–1100, 1997.

[39] P. M. Pardalos, T. Qian, et M. G. Resende, "A greedy randomized adaptive search procedure for the feedback vertex set problem", *Journal of Combinatorial Optimization*, vol. 2, no. 4, pp. 399–412, 1998.

[40] C.-M. Pintea, C. Chira, et D. Dumitrescu, "New results of ant algorithms for the linear ordering problem", *arXiv preprint arXiv :1208.5340*, 2012.

[41] C. S. Sakuraba et M. Yagiura, "Efficient local search algorithms for the linear ordering problem", *International Transactions in Operational Research*, vol. 17, no. 6, pp. 711–737, 2010.

[42] C. S. Sakuraba, D. P. Ronconi, E. G. Birgin, et M. Yagiura, "Metaheuristics for large-scale instances of the linear ordering problem", *Expert Systems with Applications*, vol. 42, no. 9, pp. 4432–4442, 2015.

[43] T. Schiavinotto et T. Stützle, "The linear ordering problem : Instances, search space analysis and algorithms", *Journal of Mathematical Modelling and Algorithms*, vol. 3, no. 4, pp. 367–402, 2004.

[44] G. C. Valdez et S. S. B. Medina, "Iterated local search for the linear ordering problem", *International Journal of Combinatorial Optimization Problems and Informatics*, vol. 3, no. 1, p. 12, 2012.

[45] T. Ye, T. Wang, Z. Lu, et J.-K. Hao, "A multi-parent memetic algorithm for the linear ordering problem", *arXiv preprint arXiv :1405.4507*, 2014.

ANNEXE A Time to reach a local optimum

Tables A.1 and A.2 report the average time to reach a local optimum in the context of HCR for the BI and FVI policy respectively. In each table, the execution time is presented for all problem instances in our dataset and for the three implementations described in 4.2.5.

Table A.1 Average execution time of a HC in HCR for the BI policy

n	Impl.	Density					
		1%	5%	10%	25%	50%	100%
63	reg	0.000083	0.000179	0.000191	0.000268	0.000282	0.000371
	reg+	0.000023	0.000041	0.000053	0.000120	0.000207	0.000479
	tree	0.000041	0.000058	0.000074	0.000190	0.000362	0.000873
125	reg	0.000731	0.001136	0.001533	0.001769	0.00202	0.00224
	reg+	0.000076	0.000141	0.000277	0.000601	0.00126	0.00259
	tree	0.000099	0.000161	0.000324	0.000807	0.00190	0.00399
250	reg	0.005310	0.010148	0.01186	0.01439	0.01616	0.0178
	reg+	0.000253	0.000891	0.00163	0.00422	0.00888	0.0205
	tree	0.000265	0.000813	0.00151	0.00428	0.01062	0.0229
500	reg	0.05197	0.08866	0.10462	0.1236	0.1391	0.150
	reg+	0.00126	0.00582	0.01186	0.0337	0.0731	0.155
	tree	0.00103	0.00372	0.00856	0.0269	0.0831	0.254
1000	reg	0.58315	0.8840	0.9963	1.269	1.199	1.29
	reg+	0.00865	0.0452	0.0991	0.283	0.612	1.32
	tree	0.00686	0.0236	0.0572	0.235	0.674	1.72
2000	reg	7.2080	10.409	11.787	13.70	15.07	16.1
	reg+	0.0865	0.516	1.156	3.34	7.32	16.2
	tree	0.0420	0.177	0.456	1.60	4.02	10.2
4000	reg	76.768	107.80	119.16	140.17	148.7	154.1
	reg+	0.834	5.35	11.84	34.89	75.5	154.1
	tree	0.248	1.15	2.82	9.05	22.6	54.9
8000	reg	716.64	961.53	1107.2	1263.4	1323	1415
	reg+	7.75	49.46	107.9	299.2	646	1418
	tree	1.41	7.12	16.3	50.7	131	323

Table A.2 Average execution time of a HC in HCR for the FVI policy

n	Impl.	Density					
		1%	5%	10%	25%	50%	100%
63	reg	0.000026	0.000027	0.000025	0.000034	0.000034	0.000050
	reg+	0.000017	0.000021	0.000022	0.000035	0.000040	0.000068
	tree	0.000041	0.000055	0.000074	0.000209	0.000445	0.001194
125	reg	0.000080	0.000076	0.000101	0.000110	0.000129	0.000181
	reg+	0.000048	0.000060	0.000087	0.000114	0.000152	0.000225
	tree	0.000093	0.000158	0.000352	0.000975	0.002501	0.005893
250	reg	0.000247	0.000381	0.000425	0.000482	0.000622	0.000809
	reg+	0.000144	0.000298	0.000363	0.000501	0.000722	0.000977
	tree	0.000250	0.000881	0.001808	0.005499	0.014780	0.033160
500	reg	0.000976	0.00188	0.00234	0.00272	0.00357	0.00345
	reg+	0.000602	0.00136	0.00193	0.00264	0.00355	0.00401
	tree	0.000875	0.00428	0.01006	0.03418	0.11961	0.40085
1000	reg	0.00741	0.01181	0.01304	0.0145	0.0145	0.0157
	reg+	0.00395	0.00710	0.00919	0.0144	0.0171	0.0200
	tree	0.00578	0.02541	0.07158	0.3285	1.0292	2.8268
2000	reg	0.0400	0.0627	0.0742	0.0889	0.106	0.122
	reg+	0.0259	0.0519	0.0679	0.0956	0.112	0.143
	tree	0.0330	0.2100	0.6096	2.3970	6.484	17.241
4000	reg	0.230	0.377	0.430	0.534	0.608	0.727
	reg+	0.155	0.323	0.406	0.554	0.634	0.808
	tree	0.197	1.510	4.077	14.529	38.465	96.608
8000	reg	1.308	2.10	2.32	2.78	3.26	4.05
	reg+	0.915	1.81	2.21	3.26	3.54	4.51
	tree	1.218	9.65	24.98	89.04	228.72	574.50

Tables A.3 and A.4 report the standard deviation of the time to reach a local optimum in the context of HCR for the BI and FVI policy respectively. In each table, the results are presented for all problem instances in our dataset up to a size of $n = 4000$ and for the three implementations described in 4.2.5.

Table A.3 Standard deviation of the execution time of a HC in HCR for the BI policy

n	Impl.	Density					
		1%	5%	10%	25%	50%	100%
63	reg	0.000011	0.000017	0.000017	0.000045	0.000018	0.000072
	reg+	0.000004	0.000004	0.000004	0.000022	0.000012	0.000112
	tree	0.000007	0.000005	0.000006	0.000034	0.000024	0.000192
125	reg	0.000101	0.000180	0.000335	0.000041	0.000061	0.000069
	reg+	0.000010	0.000025	0.000075	0.000013	0.000033	0.000303
	tree	0.000015	0.000027	0.000080	0.000012	0.000045	0.000277
250	reg	0.001184	0.001358	0.000626	0.000394	0.000355	0.000306
	reg+	0.000079	0.000276	0.000300	0.000094	0.000248	0.002894
	tree	0.000085	0.000220	0.000214	0.000093	0.000195	0.000608
500	reg	0.000424	0.000747	0.002608	0.001622	0.00140	0.00357
	reg+	0.000186	0.000453	0.000300	0.000455	0.00248	0.00430
	tree	0.000127	0.000056	0.001852	0.002222	0.00179	0.00420
1000	reg	0.051661	0.079577	0.076083	0.07288	0.01997	0.0189
	reg+	0.000076	0.000508	0.001193	0.00321	0.00491	0.0232
	tree	0.000068	0.000155	0.000749	0.00481	0.00997	0.0304
2000	reg	0.022155	0.05191	0.0693	0.1401	0.1970	0.219
	reg+	0.001072	0.01087	0.0267	0.0403	0.0607	0.459
	tree	0.000184	0.00387	0.0134	0.0265	0.0488	0.190
4000	reg	2.09459	2.94194	3.0628	2.9806	4.506	0.463
	reg+	0.02288	0.14305	0.1710	0.3838	1.681	2.942
	tree	0.00390	0.00729	0.0504	0.0722	0.147	0.453

Table A.4 Standard deviation of the execution time of a HC in HCR for the FVI policy

n	Impl.	Density					
		1%	5%	10%	25%	50%	100%
63	reg	0.000003	0.000003	0.000002	0.000006	0.000002	0.000014
	reg+	0.000002	0.000002	0.000002	0.000006	0.000003	0.000018
	tree	0.000007	0.000006	0.000005	0.000037	0.000019	0.000301
125	reg	0.000011	0.000014	0.000033	0.000006	0.000011	0.000083
	reg+	0.000007	0.000010	0.000029	0.000006	0.000010	0.000091
	tree	0.000015	0.000027	0.000102	0.000023	0.000055	0.001528
250	reg	0.000088	0.000158	0.000146	0.000022	0.000202	0.000313
	reg+	0.000048	0.000116	0.000109	0.000022	0.000211	0.000322
	tree	0.000090	0.000306	0.000454	0.000090	0.001484	0.001864
500	reg	0.000227	0.000625	0.000770	0.000817	0.001172	0.000866
	reg+	0.000128	0.000366	0.000508	0.000525	0.000780	0.000634
	tree	0.000162	0.000779	0.001107	0.000827	0.002285	0.010447
1000	reg	0.001457	0.001494	0.001533	0.00278	0.001580	0.000814
	reg+	0.000485	0.000489	0.000256	0.00306	0.000878	0.000998
	tree	0.000267	0.000242	0.002246	0.00579	0.033045	0.060550
2000	reg	0.001539	0.00491	0.00281	0.00569	0.00263	0.00568
	reg+	0.000237	0.00261	0.00177	0.00243	0.00727	0.00286
	tree	0.000207	0.00727	0.02550	0.08433	0.11617	0.24184
4000	reg	0.00953	0.00684	0.0180	0.0285	0.00800	0.0134
	reg+	0.00486	0.01358	0.0180	0.0133	0.03211	0.0308
	tree	0.00448	0.06313	0.1921	0.1949	0.21522	0.6632